

Data Science in Production: Building Scalable Model Pipelines with Python

Contents

Preface	vii
0.1 Prerequisites	vii
0.2 Book Contents	viii
0.3 Code Examples	x
0.4 Acknowledgements	x
 1 Introduction	 1
1.1 Applied Data Science	3
1.2 Python for Scalable Compute	4
1.3 Cloud Environments	6
1.3.1 Amazon Web Services (AWS)	7
1.3.2 Google Cloud Platform (GCP)	8
1.4 Coding Environments	9
1.4.1 Jupyter on EC2	9
1.5 Datasets	13
1.5.1 BigQuery to Pandas	15
1.5.2 Kaggle to Pandas	18
1.6 Prototype Models	19
1.6.1 Linear Regression	20
1.6.2 Logistic Regression	21
1.6.3 Keras Regression	22
1.7 Automated Feature Engineering	26
1.8 Conclusion	31
 2 Models as Web Endpoints	 33
2.1 Web Services	34
2.1.1 Echo Service	35
2.2 Model Persistence	40
2.2.1 Scikit-Learn	40
2.2.2 Keras	42

2.3	Model Endpoints	43
2.3.1	Scikit-Learn	44
2.3.2	Keras	46
2.4	Deploying a Web Endpoint	48
2.4.1	Gunicorn	48
2.4.2	Heroku	49
2.5	Interactive Web Services	52
2.5.1	Dash	52
2.6	Conclusion	56
3	Models as Serverless Functions	57
3.1	Managed Services	58
3.2	Cloud Functions (GCP)	59
3.2.1	Echo Service	60
3.2.2	Cloud Storage (GCS)	64
3.2.3	Model Function	66
3.2.4	Keras Model	70
3.2.5	Access Control	72
3.2.6	Model Refreshes	73
3.3	Lambda Functions (AWS)	74
3.3.1	Echo Function	74
3.3.2	Simple Storage Service (S3)	76
3.3.3	Model Function	78
3.4	Conclusion	85
4	Containers for Reproducible Models	87
4.1	Docker	88
4.2	Orchestration	92
4.2.1	AWS Container Registry (ECR)	93
4.2.2	AWS Container Service (ECS)	97
4.2.3	Load Balancing	102
4.3	Kubernetes on GCP	104
4.4	Conclusion	107
5	Workflow Tools for Model Pipelines	109
5.1	Sklearn Workflow	110
5.2	Cron	116
5.2.1	Cloud Cron	118

5.3	Workflow Tools	120
5.3.1	Apache Airflow	121
5.3.2	Managed Airflow	125
5.4	Conclusion	127
6	PySpark for Batch Pipelines	129
6.1	Spark Environments	131
6.1.1	Spark Clusters	132
6.1.2	Databricks Community Edition	133
6.2	Staging Data	136
6.2.1	S3 Credentials	137
6.3	A PySpark Primer	139
6.3.1	Persisting Dataframes	140
6.3.2	Converting Dataframes	143
6.3.3	Transforming Data	145
6.3.4	Pandas UDFs	150
6.3.5	Best Practices	154
6.4	MLlib Batch Pipeline	155
6.4.1	Vector Columns	157
6.4.2	Model Application	157
6.5	Distributed Deep Learning	161
6.5.1	Model Training	162
6.5.2	Model Application	163
6.6	Distributed Feature Engineering	166
6.6.1	Feature Generation	167
6.6.2	Feature Application	169
6.7	GCP Model Pipeline	171
6.7.1	BigQuery Export	171
6.7.2	GCP Credentials	172
6.7.3	Model Pipeline	174
6.8	Productizing PySpark	178
6.9	Conclusion	179
7	Cloud Dataflow for Batch Modeling	181
7.1	Apache Beam	183
7.2	Batch Model Pipeline	188
7.2.1	Model Training	188

7.2.2	BigQuery Publish	190
7.2.3	Datastore Publish	196
7.3	Conclusion	199
8	Streaming Model Workflows	201
8.1	Spark Streaming	202
8.1.1	Apache Kafka	203
8.1.2	Sklearn Streaming	206
8.2	Dataflow Streaming	213
8.2.1	PubSub	214
8.2.2	Natality Streaming	216
8.3	Conclusion	221
8.4	Thank You	222

Preface

This book was developed using the leanpub¹ platform. Please send any feedback or corrections to: bgweber@gmail.com

The data science landscape is constantly evolving, because new tools and libraries are enabling smaller teams to deliver more impactful products. In the current state, data scientists are expected to build systems that not only scale to a single product, but a portfolio of products. The goal of this book is to provide data scientists with a set of tools that can be used to build predictive model services for product teams.

This text is meant to be a Data Science 201 course for data science practitioners that want to develop skills for the applied science discipline. The target audience is readers with past experience with Python and scikit-learn than want to learn how to build data products. The goal is to get readers hands-on with a number of tools and cloud environments that they would use in industry settings.

0.1 Prerequisites

This book assumes that readers have prior knowledge of Python and Pandas, as well as some experience with modeling packages such as scikit-learn. This is a book that will focus on breadth rather than depth, where the goal is to get readers hands on with a number of different tools.

Python has a large library of books available, covering the language fundamentals, specific packages, and disciplines such as data

¹<https://leanpub.com/ProductionDataScience>

science. Here are some of the books I would recommend for readers to build additional knowledge of the Python ecosystem.

- **Python And Pandas**

- *Data Science from Scratch* (Grus, 2015): Introduces Python from a data science perspective.
- *Python for Data Analysis* (McKinney, 2017): Provides extensive details on the Pandas library.

- **Machine Learning**

- *Hands-On Machine Learning* (Géron, 2017): Covers scikit-learn in depth as well as TensorFlow and Keras.
- *Deep Learning for Python* (Chollet, 2017): Provides an excellent introduction to deep learning concepts using Keras as the core framework.

I will walk through the code samples in this book in detail, but will not cover the fundamentals of Python. Readers may find it useful to first explore these texts before digging into building large scale pipelines in Python.

0.2 Book Contents

The general theme of the book is to take simple machine learning models and to scale them up in different configurations across multiple cloud environments. Here's the topics covered in this book:

1. **Introduction:** This chapter will motivate the use of Python and discuss the discipline of applied data science, present the data sets, models, and cloud environments used throughout the book, and provide an overview of automated feature engineering.
2. **Models as Web Endpoints:** This chapter shows how to use web endpoints for consuming data and hosting machine learning models as endpoints using the Flask and Gunicorn libraries. We'll start with scikit-learn models and also set up a deep learning endpoint with Keras.

3. **Models as Serverless Functions:** This chapter will build upon the previous chapter and show how to set up model endpoints as serverless functions using AWS Lambda and GCP Cloud Functions.
4. **Containers for Reproducible Models:** This chapter will show how to use containers for deploying models with Docker. We'll also explore scaling up with ECS and Kubernetes, and building web applications with Plotly Dash.
5. **Workflow Tools for Model Pipelines:** This chapter focuses on scheduling automated workflows using Apache Airflow. We'll set up a model that pulls data from BigQuery, applies a model, and saves the results.
6. **PySpark for Batch Modeling:** This chapter will introduce readers to PySpark using the community edition of Databricks. We'll build a batch model pipeline that pulls data from a data lake, generates features, applies a model, and stores the results to a No SQL database.
7. **Cloud Dataflow for Batch Modeling:** This chapter will introduce the core components of Cloud Dataflow and implement a batch model pipeline for reading data from BigQuery, applying an ML model, and saving the results to Cloud Datastore.
8. **Streaming Model Workflows:** This chapter will introduce readers to Kafka and PubSub for streaming messages in a cloud environment. After working through this material, readers will learn how to use these message brokers to creating streaming model pipelines with PySpark and Dataflow that provide near real-time predictions.

After working through this material, readers should have hands-on experience with many of the tools needed to build data products, and have a better understanding of how to build scalable machine learning pipelines in a cloud environment.

0.3 Code Examples

Since the focus of this book is to get readers hands on with Python code, I have provided code listings for a subset of the chapters on GitHub. The following URL provides listings for code examples that work well in a Jupyter environment:

- https://github.com/bgweber/DS_Production

Due to formatting restrictions, many of the code snippets in this book break commands into multiple lines while omitting the continuation operator (`\`). To get code blocks to work in Jupyter or another Python coding environment, you may need to remove these line breaks. The code samples in the notebooks listed above do not add these line breaks and can be executed without modification, excluding credential and IP changes. This book uses the terms `scikit-learn` and `sklearn` interchangeably with `sklearn` used explicitly in Section 3.3.3.

0.4 Acknowledgements

I was able to author this book using Yihui Xie's excellent book-down package (Xie, 2015). For the design, I used Shashi Kumar's template² available under the Creative Commons 4.0 license. The book cover uses Cédric Franchetti's image from pxhere³.

This book was last updated on December 31, 2019.

²<https://bit.ly/2MjFDgV>

³<https://pxhere.com/en/photo/1417846>

1

Introduction

Putting predictive models into production is one of the most direct ways that data scientists can add value to an organization. By learning how to build and deploy scalable model pipelines, data scientists can own more of the model production process and rapidly deliver data products. Building data products is more than just putting code into production, it also includes DevOps and lifecycle management of live systems.

Throughout this book, we'll cover different cloud environments and tools for building scalable data and model pipelines. The goal is to provide readers with the opportunity to get hands on and start building experience with a number of different tools. While this book is targeted at analytics practitioners with prior Python experience, we'll walk through examples from start to finish, but won't dig into the details of the programming language itself.

The role of data science is constantly transforming and adding new specializations. Data scientists that build production-grade services are often called applied scientists. Their goal is to build systems that are scalable and robust. In order to be scalable, we need to use tools that can parallelize and distribute code. Parallelizing code means that we can perform multiple tasks simultaneously, and distributing code means that we can scale up the number of machines needed in order to accomplish a task. Robust services are systems that are resilient and can recover from failure. While the focus of this book is on scalability rather than robustness, we will cover monitoring systems in production and discuss measuring model performance over time.

During my career as a data scientist, I've worked at a number of video game companies and have had experience putting propensity

models, lifetime-value predictions, and recommendation systems into production. Overall, this process has become more streamlined with the development of tools such as PySpark, which enable data scientists to more rapidly build end-to-end products. While many companies now have engineering teams with machine learning focuses, it's valuable for data scientists to have broad expertise in productizing models. Owning more of the process means that a data science team can deliver products quicker and iterate much more rapidly.

Data products are useful for organizations, because they can provide personalization for the user base. For example, the recommendation system that I designed for EverQuest Landmark¹ provided curated content for players from a marketplace with thousands of user-created items. The goal of any data product should be creating value for an organization. The recommendation system accomplished this goal by increasing the revenue generated from user-created content. Propensity models, which predict the likelihood of a user to perform an action, can also have a direct impact on core metrics for an organization, by enabling personalized experiences that increase user engagement.

The process used to productize models is usually unique for each organization, because of different cloud environments, databases, and product organizations. However, many of the same tools are used within these workflows, such as SQL and PySpark. Your organization may not be using the same data ecosystem as these examples, but the methods should transfer to your use cases.

In this chapter, we will introduce the role of applied science and motivate the usage of Python for building data products, discuss different cloud and coding environments for scaling up data science, introduce the data sets and types of models used throughout the book, and introduce automated feature engineering as a step to include in data science workflows.

¹<https://bit.ly/2YFLYPg>

1.1 Applied Data Science

Data science is a broad discipline with many different specializations. One distinction that is becoming common is product data science and applied data science. Product data scientists are typically embedded on a product team, such as a game studio, and provide analysis and modeling that helps the team directly improve the product. For example, a product data scientist might find an issue with the first-time user experience in a game, and make recommendations such as which languages to focus on for localization to improve new user retention.

Applied science is at the intersection of machine learning engineering and data science. Applied data scientists focus on building data products that product teams can integrate. For example, an applied scientist at a game publisher might build a recommendation service that different game teams can integrate into their products. Typically, this role is part of a central team that is responsible for owning a data product. A data product is a production system that provides predictive models, such as identifying which items a player is likely to buy.

Applied scientist is a job title that is growing in usage across tech companies including Amazon, Facebook, and Microsoft. The need for this type of role is growing, because a single applied scientist can provide tremendous value to an organization. For example, instead of having product data scientists build bespoke propensity models for individual games, an applied scientist can build a scalable approach that provides a similar service across a portfolio of games. At Zynga, one of the data products that the applied data science team built was a system called AutoModel², which provided several propensity models for all of our games, such as the likelihood for a specific player to churn.

There's been a few developments in technology that have made applied science a reality. Tools for automated feature engineering,

²<https://ubm.io/2KdYRDq>

such as deep learning, and scalable computing environments, such as PySpark, have enabled companies to build large scale data products with smaller team sizes. Instead of hiring engineers for data ingestion and warehousing, data scientists for predictive modeling, and additional engineers for building a machine learning infrastructure, you can now use managed services in the cloud to enable applied scientists to take on more of the responsibilities previously designated to engineering teams.

One of the goals of this book is to help data scientists make the transition to applied science, by providing hands-on experience with different tools that can be used for scalable compute and standing up services for predictive models. We will work through different tools and cloud environments to build proof of concepts for data products that can translate to production environments.

1.2 Python for Scalable Compute

Python is quickly becoming the de facto language for data science. In addition to the huge library of packages that provide useful functionality, one of the reasons that the language is becoming so popular is that it can be used for building scalable data and predictive model pipelines. You can use Python on your local machine and build predictive models with scikit-learn, or use environments such as Dataflow and PySpark to build distributed systems. While these different environments use different libraries and programming paradigms, it's all in the same language of Python. It's no longer necessary to translate an R script into a production language such as Java, you can use the same language for both development and production of predictive models.

It took me awhile to adopt Python as my data science language of choice. Java had been my preferred language, regardless of task, since early in my undergraduate career. For data science tasks, I used tools like Weka to train predictive models. I still find Java to be useful when building data pipelines, and it's great to know in

order to directly collaborate with engineering teams on projects. I later switched to R while working at Electronic Arts, and found the transition to an interactive coding environment to be quite useful for data science. One of the features I really enjoyed in R is R Markdown, which you can use to write documents with inline code. In fact, this entire book was written using an extension of R Markdown called bookdown (Xie, 2019). I later switched to using R within Jupyter notebooks and even wrote a book on using R and Java for data science at a startup (Weber, 2018).

When I started working at Zynga in 2018, I adopted Python and haven't looked back. It took a bit of time to get used to the new language, but there are a number of reasons that I wanted to learn Python:

- **Momentum:** Many teams are already using Python for production, or portions of their data pipelines. It makes sense to also use Python for performing analysis tasks.
- **PySpark:** R and Java don't provide a good transition to authoring Spark tasks interactively. You can use Java for Spark, but it's not a good fit for exploratory work, and the transition from Python to PySpark seems to be the most approachable way to learn Spark.
- **Deep Learning:** I'm interested in deep learning, and while there are R bindings for libraries such as Keras, it's better to code in the native language of these libraries. I previously used R to author custom loss functions, and debugging errors was problematic.
- **Libraries:** In addition to the deep learning libraries offered for Python, there's a number of other useful tools including Flask and Bokeh. There's also notebook environments that can scale including Google's Colaboratory and AWS SageMaker.

To ease the transition from R to Python, I took the following steps:

- **Focus on outcomes, not semantics:** Instead of learning about all of the fundamentals of the language, I first focused on doing in Python what I already knew how to do in other languages, such as training a logistic regression model.

- **Learn the ecosystem, not the language:** I didn't limit myself to the base language when learning, and instead jumped right in to using Pandas and scikit-learn.
- **Use cross-language libraries:** I was already familiar working with Keras and Plotly in R, and used knowledge of these libraries to bootstrap learning Python.
- **Work with real-world data:** I used the data sets provided by Google's BigQuery to test out my scripts on large-scale data.
- **Start locally if possible:** While one of my goals was to learn PySpark, I first focused on getting things up and running on my local machine before moving to cloud ecosystems.

There are many situations where Python is not the best choice for a specific task, but it does have broad applicability when prototyping models and building scalable model pipelines. Because of Python's rich ecosystem, we will be using it for all of the examples in this book.

1.3 Cloud Environments

In order to build scalable data science pipelines, it's necessary to move beyond single machine scripts and move to clusters of machines. While this is possible to do with an on-premise setup, a common trend is using cloud computing environments in order to achieve large-scale processing. There's a number of different options available, with the top three platforms currently being Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

Most cloud platforms offer free credits for getting started. GCP offers a \$300 credit for new users to get hands on with their tools, while AWS provides free-tier access for many services. In this book, we'll get hands on with both AWS and GCP, with little to no cost involved.

1.3.1 Amazon Web Services (AWS)

AWS is currently the largest cloud provider, and this dominance has been demonstrated by the number of gaming companies using the platform. I've had experience working with AWS at Electronic Arts, Twitch, and Zynga. This platform has a wide range of tools available, but getting these components to work well together generally takes additional engineering work versus GCP.

With AWS, you can use both self-hosted and managed solutions for building data pipelines. For example, the managed option for messaging on AWS is Kinesis, while the self-hosted option is Kafka. We'll walk through examples with both of these options in Chapter 8. There's typically a tradeoff between cost and DevOps when choosing between self-hosted and managed options.

The default database to use on AWS is Redshift, which is a columnar database. This option works well as a data warehouse, but it doesn't scale well to data lake volumes. It's common for organizations to set up a separate data lake and data warehouse on AWS. For example, it's possible to store data on S3 and use tools such as Athena to provide data lake functionality, while using Redshift as a solution for a data warehouse. This approach has worked well in the past, but it creates issues when building large-scale data science pipelines. Moving data in and out of a relational database can be a bottleneck for these types of workflows. One of the solutions to this bottleneck is to use vendor solutions that separate storage from compute, such as Snowflake or Delta Lake.

The first component we'll work with in AWS is Elastic Compute (EC2) instances. These are individual virtual machines that you can spin up and provision for any necessary task. In section 1.4.1, we'll show how to set up an instance that provides a remote Jupyter environment. EC2 instances are great for getting started with tools such as Flask and Gunicorn, and getting started with Docker. To scale up beyond individual instances, we'll explore Lambda functions and Elastic Container Services.

To build scalable pipelines on AWS, we'll focus on PySpark as the main environment. PySpark enables Python code to be distributed

across a cluster of machines, and vendors such as Databricks provide managed environments for Spark. Another option that is available only on AWS is SageMaker, which provides a Jupyter notebook environment for training and deploying models. We are not covering SageMaker in this book, because it is specific to AWS and currently supports only a subset of predictive models. Instead, we'll explore tools such as MLflow.

1.3.2 Google Cloud Platform (GCP)

GCP is currently the third largest cloud platform provider, and offers a wide range of managed tools. It's currently being used by large media companies such as Spotify, and within the games industry being used by King and Niantic. One of the main benefits of using GCP is that many of the components can be wired together using Dataflow, which is a tool for building batch and streaming data pipelines. We'll create a batch model pipeline with Dataflow in Chapter 7 and a streaming pipeline in Chapter 8.

Google Cloud Platform currently offers a smaller set of tools than AWS, but there is feature parity for many common tools, such as PubSub in place of Kinesis, and Cloud Functions in place of AWS Lambda. One area where GCP provides an advantage is BigQuery as a database solution. BigQuery separates storage from compute, and can scale to both data lake and data warehouse use cases.

Dataflow is one of the most powerful tools for data scientists that GCP provides, because it empowers a single data scientist to build large-scale pipelines with much less effort than other platforms. It enables building streaming pipelines that connect PubSub for messaging, BigQuery for analytics data stores, and BigTable for application databases. It's also a managed solution that can autoscale to meet demand. While the original version of Dataflow was specific to GCP, it's now based on the Apache Beam library which is portable to other platforms.

1.4 Coding Environments

There's a variety of options for writing Python code in order to do data science. The best environment to use likely varies based on what you are building, but notebook environments are becoming more and more common as the place to write Python scripts. The three types of coding environments I've worked with for Python are IDEs, text editors, and notebooks.

If you're used to working with an IDE, tools like PyCharm and Rodeo are useful editors and provide additional tools for debugging versus other options. It's also possible to write code in text editors such as Sublime and then run scripts via the command line. I find this works well for building web applications with Flask and Dash, where you need to have a long running script that persists beyond the scope of running a cell in a notebook. I now perform the majority of my data science work in notebook environments, and this covers exploratory analysis and productizing models.

I like to work in coding environments that make it trivial to share code and collaborate on projects. Databricks and Google Colab are two coding environments that provide truly collaborative notebooks, where multiple data scientists can simultaneously work on a script. When using Jupyter notebooks, this level of real-time collaboration is not currently supported, but it's good practice to share notebooks in version control systems such as GitHub for sharing work.

In this book, we'll use only the text editor and notebook environments for coding. For learning how to build scalable pipelines, I recommend working on a remote machine, such as EC2, to become more familiar with cloud environments, and to build experience setting up Python environments outside of your local machine.

1.4.1 Jupyter on EC2

To get experience with setting up a remote machine, we'll start by setting up a Jupyter notebook environment on a EC2 instance in

Instance ID	i-0b3e8052d00a567d9	Public DNS (IPv4)	ec2-54-87-230-152.compute-1.amazonaws.com
Instance state	running	IPv4 Public IP	54.87.230.152
Instance type	t2.micro	IPv6 IPs	-
Elastic IPs		Private DNS	ip-172-31-53-82.ec2.internal
Availability zone	us-east-1a	Private IPs	172.31.53.82
Security groups	launch-wizard-66 , view inbound rules , view outbound rules	Secondary private IPs	

FIGURE 1.1: Public and Private IPs on EC2.

AWS. The result is a remote machine that we can use for Python scripting. Accomplishing this task requires spinning up an EC2 instance, configuring firewall settings for the EC2 instance, connecting to the instance using SSH, and running a few commands to deploy a Jupyter environment on the machine.

The first step is to set up an AWS account and log into the AWS management console. AWS provides a free account option with free-tier access to a number of services including EC2. Next, provision a machine using the following steps:

1. Under “Find Services”, search for EC2
2. Click “Launch Instance”
3. Select a free-tier Amazon Linux AMI
4. Click “Review and Launch”, and then “Launch”
5. Create a key pair and save to your local machine
6. Click “Launch Instances”
7. Click “View Instances”

The machine may take a few minutes to provision. Once the machine is ready, the instance state will be set to “running”. We can now connect to the machine via SSH. One note on the different AMI options is that some of the configurations are set up with Python already installed. However, this book focuses on Python 3 and the included version is often 2.7.

There’s two different IPs that you need in order to connect to the machine via SSH and later connect to the machine via web browser. The public and private IPs are listed under the “Description” tab

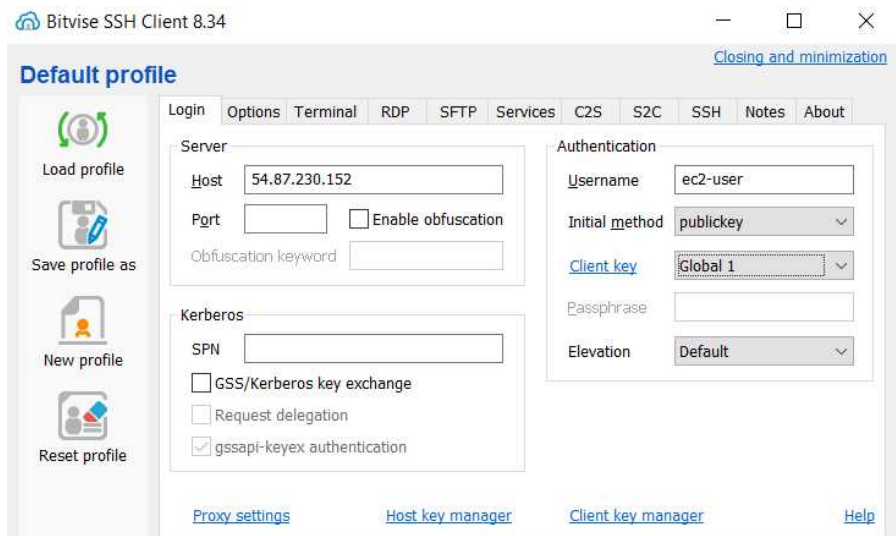


FIGURE 1.2: SSH connection settings.

as shown in Figure 1.1. To connect to the machine via SSH we'll use the Public IP (54.87.230.152). For connecting to the machine, you'll need to use an SSH client such as Putty if working in a Windows environment. For Linux and Mac OS, you can use ssh via the command line. To connect to the machine, use the user name "ec2-user" and the key pair generated when launching the instance. An example of connecting to EC2 using the Bitvise client on Windows is shown in Figure 1.2.

Once you connect to the machine, you can check the Python version by running `python --version`. On my machine, the result was 2.7.16, meaning that additional setup is needed in order to upgrade to Python 3. We can run the following commands to install Python 3, pip, and Jupyter.

```
sudo yum install -y python37
python3 --version
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
sudo python3 get-pip.py
```

```
pip --version  
pip install --user jupyter
```

The two version commands are to confirm that the machine is pointing at Python 3 for both Python and pip. Once Jupyter is installed, we'll need to set up a firewall restriction so that we can connect directly to the machine on port 8888, where Jupyter runs by default. This approach is the quickest way to get connected to the machine, but it's advised to use SSH tunneling to connect to the machine rather than a direct connection over the open web. You can open up port 8888 for your local machine by performing the following steps from the EC2 console:

1. Select your EC2 instance
2. Under "Description", select security groups
3. Click "Actions" -> "Edit Inbound Rules"
4. Add a new rule: change the port to 8888, select "My IP"
5. Click "Save"

We can now run and connect to Jupyter on the EC2 machine. To launch Jupyter, run the command shown below while replacing the IP with your EC2 instance's Private IP. It is necessary to specify the `--ip` parameter in order to enable remote connections to Jupyter, as incoming traffic will be routed via the private IP.

```
jupyter notebook --ip 172.31.53.82
```

When you run the `jupyter notebook` command, you'll get a URL with a token that can be used to connect to the machine. Before entering the URL into your browser, you'll need to swap the Private IP output to the console with the Public IP of the EC2 instance, as shown in the snippet below.

```
# Original URL
```

The Jupyter Notebook is running at:



FIGURE 1.3: Jupyter Notebook on EC2.

```
http://172.31.53.82:8888/?token=
98175f620fd68660d26fa7970509c6c49ec2afc280956a26

# Swap Private IP with Public IP
http://54.87.230.152:8888/?token=
98175f620fd68660d26fa7970509c6c49ec2afc280956a26
```

You can now paste the updated URL into your browser to connect to Jupyter on the EC2 machine. The result should be a Jupyter notebook fresh install with a single file `get-pip.py` in the base directory, as shown in Figure 1.3. Now that we have a machine set up with Python 3 and Jupyter notebook, we can start exploring different data sets for data science.

1.5 Datasets

To build scalable data pipelines, we'll need to switch from using local files, such as CSVs, to distributed data sources, such as Parquet files on S3. While the tools used across cloud platforms to load data vary significantly, the end result is usually the same, which is a dataframe. In a single machine environment, we can use Pandas to load the dataframe, while distributed environments use different implementations such as Spark dataframes in PySpark.

This section will introduce the data sets that we'll explore throughout the rest of the book. In this chapter we'll focus on loading the

data using a single machine, while later chapters will present distributed approaches. While most of the data sets presented here can be downloaded as CSV files and read into Pandas using `read_csv`, it's good practice to develop automated workflows to connect to diverse data sources. We'll explore the following datasets throughout this book:

- **Boston Housing:** Records of sale prices of homes in the Boston housing market back in 1980.
- **Game Purchases:** A synthetic data set representing games purchased by different users on Xbox One.
- **Natality:** One of BigQuery's open data sets on birth statistics in the US over multiple decades.
- **Kaggle NHL:** Play-by-play events from professional hockey games and game statistics over the past decade.

The first two data sets are single commands to load, as long as you have the required libraries installed. The Natality and Kaggle NHL data sets require setting up authentication files before you can programmatically pull the data sources into Pandas.

The first approach we'll use to load a data set is to retrieve it directly from a library. Multiple libraries include the Boston housing data set, because it is a small data set that is useful for testing out regression models. We'll load it from scikit-learn by first running pip from the command line:

```
pip install --user pandas
pip install --user sklearn
```

Once scikit-learn is installed, we can switch back to the Jupyter notebook to explore the data set. The code snippet below shows how to load the scikit-learn and Pandas libraries, load the Boston data set as a Pandas dataframe, and display the first 5 records. The result of running these commands is shown in Figure 1.4.

```
from sklearn.datasets import load_boston
import pandas as pd
```


CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	label
0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

FIGURE 1.4: Boston Housing data set.

```
data, target = load_boston(True)
bostonDF = pd.DataFrame(data, columns=load_boston().feature_names)
bostonDF['label'] = target
bostonDF.head()
```

The second approach we'll use to load a data set is to fetch it from the web. The CSV for the Games data set is available as a single file on GitHub. We can fetch it into a Pandas dataframe by using the `read_csv` function and passing the URL of the file as a parameter. The result of reading the data set and printing out the first few records is shown in Figure 1.5.

```
gamesDF = pd.read_csv("https://github.com/bgweber/
    Twitch/raw/master/Recommendations/games-expand.csv")
gamesDF.head()
```

Both of these approaches are similar to downloading CSV files and reading them from a local directory, but by using these methods we can avoid the manual step of downloading files.

1.5.1 BigQuery to Pandas

One of the ways to automate workflows authored in Python is to directly connect to data sources. For databases, you can use connectors based on JDBC or native connectors, such as the `bigquery` module provided by the Google Cloud library. This connector enables Python applications to send queries to BigQuery and load the results as a Pandas dataframe. This process involves setting up

G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	label
0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	1
0	0	1	0	1	1	0	1	1	0	1

FIGURE 1.5: Game Purchases data set.

a GCP project, installing the prerequisite Python libraries, setting up the Google Cloud command line tools, creating GCP credentials, and finally sending queries to BigQuery programmatically.

If you do not already have a GCP account set up, you'll need to create a new account³. Google provides a \$300 credit for getting up and running with the platform. The first step is to install the Google Cloud library by running the following steps:

```
pip install --user google-cloud-bigquery
pip install --user matplotlib
```

Next, we'll need to set up the Google Cloud command line tools, in order to set up credentials for connecting to BigQuery. While the files to use will vary based on the current release⁴, here are the steps I ran on the command line:

```
curl -O https://dl.google.com/dl/cloudsdk/channels/
    rapid/downloads/google-cloud-sdk-255.0.0-
    linux-x86_64.tar.gz
tar zxvf google-cloud-sdk-255.0.0-linux-x86_64.tar.gz
    google-cloud-sdk
./google-cloud-sdk/install.sh
```

³<https://cloud.google.com/gcp>

⁴<https://cloud.google.com/sdk/install>

Once the Google Cloud command line tools are installed, we can set up credentials for connecting to BigQuery:

```
gcloud config set project project_name
gcloud auth login
gcloud init
gcloud iam service-accounts create dsdemo
gcloud projects add-iam-policy-binding your_project_id
  --member "serviceAccount:dsdemo@your_project_id.iam.
           gserviceaccount.com" --role "roles/owner"
gcloud iam service-accounts keys
  create dsdemo.json --iam-account
  dsdemo@your_project_id.iam.gserviceaccount.com
export GOOGLE_APPLICATION_CREDENTIALS=
  /home/ec2-user/dsdemo.json
```

You'll need to substitute `project_name` with your project name, `your_project_id` with your project ID, and `dsdemo` with your desired service account name. The result is a json file with credentials for the service account. The export command at the end of this process tells Google Cloud where to find the credentials file.

Setting up credentials for Google Cloud is involved, but generally only needs to be performed once. Now that credentials are configured, it's possible to directly query BigQuery from a Python script. The snippet below shows how to load a BigQuery client, send a SQL query to retrieve 10 rows from the natality data set, and pull the results into a Pandas dataframe. The resulting dataframe is shown in Figure 1.6.

```
from google.cloud import bigquery
client = bigquery.Client()
sql = """
  SELECT *
  FROM `bigquery-public-data.samples.natality`
  limit 10
  """
```

	source_year	year	month	day	wday	state	is_male	child_race	weight_pounds	plurality
0	1970	1970	9	4.0	NaN	HI	True	7.0	7.625790	NaN
1	1971	1971	6	2.0	NaN	HI	False	6.0	7.438397	1.0
2	1972	1972	11	27.0	NaN	HI	False	7.0	8.437091	1.0
3	1972	1972	11	10.0	NaN	HI	True	7.0	7.374463	1.0
4	1973	1973	12	26.0	NaN	HI	False	7.0	5.813590	1.0

5 rows × 31 columns

FIGURE 1.6: Previewing the BigQuery data set.

```
natalityDF = client.query(sql).to_dataframe()
natalityDF.head()
```

1.5.2 Kaggle to Pandas

Kaggle is a data science website that provides thousands of open data sets to explore. While it is not possible to pull Kaggle data sets directly into Pandas dataframes, we can use the Kaggle library to programmatically download CSV files as part of an automated workflow.

The quickest way to get set up with this approach is to create an account on Kaggle⁵. Next, go to the account tab of your profile and select ‘Create API Token’, download and open the file, and then run `vi .kaggle/kaggle.json` on your EC2 instance to copy over the contents to your remote machine. The result is a credential file you can use to programmatically download data sets. We’ll explore the NHL (Hockey) data set by running the following commands:

```
pip install kaggle --user

kaggle datasets download martinellis/nhl-game-data
unzip nhl-game-data.zip
chmod 0600 *.csv
```

⁵<https://www.kaggle.com>

	game_id	season	type	date_time	date_time_GMT	away_team_id	home_team_id	away_goals	home_goals
0	2011030221	20112012	P	2012-04-29	2012-04-29T19:00:00Z	1	4	3	4
1	2011030222	20112012	P	2012-05-01	2012-05-01T23:30:00Z	1	4	4	1
2	2011030223	20112012	P	2012-05-03	2012-05-03T23:30:00Z	4	1	3	4
3	2011030224	20112012	P	2012-05-06	2012-05-06T23:30:00Z	4	1	2	4
4	2011030225	20112012	P	2012-05-08	2012-05-08T23:30:00Z	1	4	3	1

FIGURE 1.7: NHL Kaggle data set.

These commands will download the data set, unzip the files into the current directory, and enable read access on the files. Now that the files are downloaded on the EC2 instance, we can load and display the Game data set, as shown in Figure 1.7. This data set includes different files, where the `game` file provides game-level summaries and the `game_plays` file provides play-by-play details.

```
import pandas as pd
nhlDF = pd.read_csv('game.csv')
nhlDF.head()
```

We walked through a few different methods for loading data sets into a Pandas dataframe. The common theme with these different approaches is that we want to avoid manual steps in our workflows, in order to automate pipelines.

1.6 Prototype Models

Machine learning is one of the most important steps in the pipeline of a data product. We can use predictive models to identify which users are most likely to purchase an item, or which users are most likely to stop using a product. The goal of this section is to present simple versions of predictive models that we'll later scale up in more complex pipelines. This book will not focus on state-of-the-

art models, but instead cover tools that can be applied to a variety of different machine learning algorithms.

The library to use for implementing different models will vary based on the cloud platform and execution environment being used to deploy a model. The regression models presented in this section are built with scikit-learn, while the models we'll build out with PySpark use MLlib.

1.6.1 Linear Regression

Regression is a common task for supervised learning, such as predicting the value of a home, and linear regression is a useful algorithm for making predictions for these types of problems. scikit-learn provides both linear and logistic regression models for making predictions. We'll start by using the `LinearRegression` class in scikit-learn to predict home prices for the Boston housing data set. The code snippet below shows how to split the Boston data set into different training and testing data sets and separate data (`train_x`) and label (`train_y`) objects, create and fit a linear regression model, and calculate error metrics on the test data set.

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# See Section 1.4 (Boston Hosing data set)
bostonDF = ...

x_train, x_test, y_train, y_test = train_test_split(
    bostonDF.drop(['label'],axis=1),bostonDF['label'],test_size=0.3)

model = LinearRegression()
model.fit(x_train, y_train)

print("R^2: " + str(model.score(x_test, y_test)))
print("Mean Error: " + str(sum(
    abs(y_test - model.predict(x_test) ))/y_test.count()))
```

The `train_test_split` function is used to split up the data set into 70% train and 30% holdout data sets. The first parameter is the data attributes from the Boston dataframe, with the label dropped, and the second parameter is the labels from the dataframe. The two commands at the end of the script calculate the R-squared value based on Pearson correlation, and the mean error is defined as the mean difference between predicted and actual home prices. The output of this script was an R^2 value of 0.699 and mean error of 3.36. Since house prices in this data set are divided by a thousand, the mean error is \$3.36k.

We now have a simple model that we can productize in a number of different environments. In later sections and chapters, we'll explore methods for scaling features, supporting more complex regression models, and automating feature generation.

1.6.2 Logistic Regression

Logistic regression is a supervised classification algorithm that is useful for predicting which users are likely to perform an action, such as purchasing a product. Using scikit-learn, the process is similar to fitting a linear regression model. The main differences from the prior script are the data set being used, the model object instantiated (`LogisticRegression`), and using the `predict_proba` function to calculate error metrics. This function predicts a probability in the continuous range of $[0,1]$ rather than a specific label. The snippet below predicts which users are likely to purchase a specific game based on prior games already purchased:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
import pandas as pd

# Games data set
gamesDF = pd.read_csv("https://github.com/bgweber/Twitch/raw/
                      master/Recommendations/games-expand.csv")
```

```
x_train, x_test, y_train, y_test = train_test_split(
    gamesDF.drop(['label'],axis=1),gamesDF['label'],test_size=0.3)

model = LogisticRegression()
model.fit(x_train, y_train)

print("Accuracy: " + str(model.score(x_test, y_test)))
print("ROC: " + str(roc_auc_score(y_test,
    model.predict_proba(x_test)[: , 1] )))
```

The output of this script is two metrics that describe the performance of the model on the holdout data set. The accuracy metric describes the number of correct predictions over the total number of predictions, and the ROC metric describes the number of correctly classified outcomes based on different model thresholds. ROC is a useful metric to use when the different classes being predicted are imbalanced, with noticeably different sizes. Since most players are unlikely to buy a specific game, ROC is a good metric to utilize for this use case. When I ran this script, the result was an accuracy of 86.6% and an ROC score of 0.757.

Linear and logistic regression models with scikit-learn are a good starting point for many machine learning projects. We'll explore more complex models in this book, but one of the general strategies I take as a data scientist is to quickly deliver a proof of concept, and then iterate and improve a model once it is shown to provide value to an organization.

1.6.3 Keras Regression

While I generally recommend starting with simple approaches when building model pipelines, deep learning is becoming a popular tool for data scientists to apply to new problems. It's great to explore this capability when tackling new problems, but scaling up deep learning in data science pipelines presents a new set of challenges. For example, PySpark does not currently have a native way of distributing the model application phase to big data.

There's plenty of books for getting started with deep learning in Python, such as (Chollet, 2017).

In this section, we'll repeat the same task from the prior section, which is predicting which users are likely to buy a game based on their prior purchases. Instead of using a shallow learning approach to predict propensity scores, we'll use the Keras framework to build a neural network for predicting this outcome. Keras is a general framework for working with deep learning implementations. We can install these dependencies from the command line:

```
pip install --user tensorflow==1.14.0
pip install --user keras==2.2.4
```

This process can take awhile to complete, and based on your environment may run into installation issues. It's recommended to verify that the installation worked by checking your Keras version in a Jupyter notebook:

```
import tensorflow as tf
import keras
from keras import models, layers
import matplotlib.pyplot as plt
keras.__version__
```

The general process for building models with Keras is to set up the structure of the model, compile the model, fit the model, and evaluate the model. We'll start with a simple model to provide a baseline, which is shown in the snippet below. This code creates a network with an input layer, a dropout layer, a hidden layer, and an output layer. The input to the model is 10 binary variables that describe prior games purchased, and the output is a prediction of the likelihood to purchase a specified game.

```
x_train, x_test, y_train, y_test = train_test_split(
    gamesDF.drop(['label'], axis=1), gamesDF['label'], test_size=0.3)
```

```
# define the network structure
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10,)))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

# define ROC AUC as a metric
def auc(y_true, y_pred):
    auc = tf.metrics.auc(y_true, y_pred)[1]
    keras.backend.get_session().run(
        tf.local_variables_initializer())
    return auc

# compile and fit the model
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy', metrics=[auc])
history = model.fit(x_train, y_train, epochs=100, batch_size=100,
                    validation_split = .2, verbose=0)
```

Since the goal is to identify the likelihood of a player to purchase a game, ROC is a good metric to use to evaluate the performance of a model. Keras does not support this directly, but we can define a custom metrics function that wraps the `auc` functionality provided by TensorFlow.

Next, we specify how to optimize the model. We'll use `rmsprop` for the optimizer and `binary_crossentropy` for the loss function. The last step is to train the model. The code snippet shows how to fit the model using the training data set, 100 training epochs with a batch size of 100, and a cross validation split of 20%. This process can take awhile to run if you increase the number of epochs or decrease the batch size. The validation set is sampled from only the training data set.

The result of this process is a history object that tracks the loss and metrics on the training and validation data sets. The code

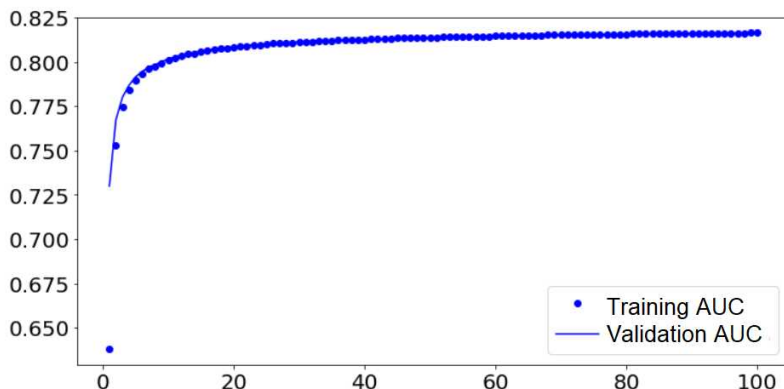


FIGURE 1.8: ROC AUC metrics for the training process.

snippet below shows how to plot these values using Matplotlib. The output of this step is shown in Figure 1.8. The plot shows that both the training and validation data sets leveled off at around a 0.82 AUC metric during the model training process. To compare this approach with the logistic regression results, we'll evaluate the performance of the model on the holdout data set.

```
loss = history.history['auc']
val_loss = history.history['val_auc']
epochs = range(1, len(loss) + 1)

plt.figure(figsize=(10,6) )
plt.plot(epochs, loss, 'bo', label='Training AUC')
plt.plot(epochs, val_loss, 'b', label='Validation AUC')
plt.legend()
plt.show()
```

To measure the performance of the model on the test data set, we can use the `evaluate` function to measure the ROC metric. The code snippet below shows how to perform this task on our training data set, which results in an ROC AUC value of 0.816. This is noticeably better than the performance of the logistic regression model, with an AUC value of 0.757, but using other shallow learning methods

such as random forests or XGBoost would likely perform much better on this task.

```
results = model.evaluate(x_test, y_test, verbose = 0)
print("ROC: " + str(results[1]))
```

1.7 Automated Feature Engineering

Automated feature engineering is a powerful tool for reducing the amount of manual work needed in order to build predictive models. Instead of a data scientist spending days or weeks coming up with the best features to describe a data set, we can use tools that approximate this process. One library I've been working with to implement this step is FeatureTools. It takes inspiration from the automated feature engineering process in deep learning, but is meant for shallow learning problems where you already have structured data, but need to translate multiple tables into a single record per user. The library can be installed as follows:

```
sudo yum install gcc
sudo yum install python3-devel
pip install --user framequery
pip install --user fsspec
pip install --user featuretools
```

In addition to this library, I loaded the framequery library, which enables writing SQL queries against dataframes. Using SQL to work with dataframes versus specific interfaces, such as Pandas, is useful when translating between different execution environments.

The task we'll apply the FeatureTools library to is predicting which games in the Kaggle NHL data set are postseason games. We'll make this prediction based on summarizations of the play events that are recorded for each game. Since there can be hundreds of play events per game, we need a process for aggregating these into

a single summary per game. Once we aggregate these events into a single game record, we can apply a logistic regression model to predict whether the game is regular or postseason.

The first step we'll perform is loading the data sets and performing some data preparation, as shown below. After loading the data sets as Pandas dataframes, we drop a few attributes from the plays object, and fill any missing attributes with 0.

```
import pandas as pd

game_df = pd.read_csv("game.csv")
plays_df = pd.read_csv("game_plays.csv")

plays_df = plays_df.drop(['secondaryType', 'periodType',
                          'dateTime', 'rink_side'], axis=1).fillna(0)
```

To translate the play events into a game summary, we'll first 1-hot encode two of the attributes in the plays dataframe, and then perform deep feature synthesis. The code snippet below shows how to perform the first step, and uses FeatureTools to accomplish this task. The result is quite similar to using the `get_dummies` function in Pandas, but this approach requires some additional steps.

The base representation in FeatureTools is an EntitySet, which describes a set of tables and the relationships between them, which is similar to defining foreign key constraints. To use the `encode_features` function, we need to first translate the plays dataframe into an entity. We can create an EntitySet directly from the `plays_df` object, but we also need to specify which attributes should be handled as categorical, using the `variable_types` dictionary parameter.

```
import featuretools as ft
from featuretools import Feature

es = ft.EntitySet(id="plays")
```

	index	event = Faceoff	event = Shot	event = Hit	event = Stoppage	event = Blocked Shot	event = Missed Shot	event = Giveaway	event = Takeaway	event = Penalty	...	team_id_against	x	y	period	periodTime	periodTimeRe
0	0	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	1	0	
1	1	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	1	0	
2	2	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	1	0	
3	3	1	0	0	0	0	0	0	0	0	...	1.0	0.0	0.0	1	0	
4	4	0	0	0	0	0	0	1	0	0	...	1.0	28.0	24.0	1	21	

5 rows x 37 columns

FIGURE 1.9: The 1-hot encoded Plays dataframe.

```
es = es.entity_from_dataframe(entity_id="plays",dataframe=plays_df
                             ,index="play_id", variable_types = {
                                 "event": ft.variable_types.Categorical,
                                 "description": ft.variable_types.Categorical })

f1 = Feature(es["plays"]["event"])
f2 = Feature(es["plays"]["description"])

encoded, defs = ft.encode_features(plays_df, [f1, f2], top_n=10)
encoded.reset_index(inplace=True)
encoded.head()
```

Next, we pass a list of features to the `encode_features` function, which returns a new dataframe with the dummy variables and a `defs` object that describes how to translate an input dataframe into the 1-hot encoded format. For pipelines later on in this book, where we need to apply transformations to new data sets, we'll store a copy of the `defs` object for later use. The result of applying this transformation to the plays dataframe is shown in Figure 1.9.

The next step is to aggregate the hundreds of play events per game into single game summaries, where the resulting dataframe has a single row per game. To accomplish this task, we'll recreate the `EntitySet` from the prior step, but use the 1-hot encoded dataframe as the input. Next, we use the `normalize_entity` function to describe games as a parent object to plays events, where all plays with the same `game_id` are grouped together. The last step is to use the `dfs` function to perform deep feature synthesis. DFS applies aggregate calculations, such as `SUM` and `MAX`, across the different features in

	game_id	SUM(plays.index)	SUM(plays.event = Faceoff)	SUM(plays.event = Shot)	SUM(plays.event = Hit)	SUM(plays.event = Stoppage)	SUM(plays.event = Blocked Shot)	SUM(plays.event = Missed Shot)	SUM(plays.event = Giveaway)
0	2010020001	1097336804	43	47	61	31	43	24	23
1	2010020002	1100541237	56	53	66	44	30	28	20
2	2010020003	1088867005	85	53	33	65	38	17	22
3	2010020004	1085722191	60	72	37	46	26	15	14
4	2010020005	1256544235	60	66	49	44	37	33	28

5 rows × 212 columns

FIGURE 1.10: Generated features for the NHL data set.

the child dataframe, in order to collapse hundreds of records into a single row.

```
es = ft.EntitySet(id="plays")
es = es.entity_from_dataframe(entity_id="plays",
                             dataframe=encoded, index="play_id")
es = es.normalize_entity(base_entity_id="plays",
                         new_entity_id="games", index="game_id")

features,transform=ft.dfs(entityset=es,
                           target_entity="games",max_depth=2)
features.reset_index(inplace=True)
features.head()
```

The result of this process is shown in Figure 1.10. The shape of the sampled dataframe, 5 rows by 212 columns, indicates that we have generated hundreds of features to describe each game using deep feature synthesis. Instead of hand coding this translation, we utilized the FeatureTools library to automate this process.

Now that we have hundreds of features for describing a game, we can use logistic regression to make predictions about the games. For this task, we want to predict whether a game is regular season or postseason, where `type = 'P'`. The code snippet below shows how to use the `framequery` library to combine the generated features with the initially loaded games dataframe using a SQL join. We use the `type` attribute to assign a label, and then return all of the generated features and the label. The result is a dataframe that we can pass to `scikit-learn`.

```
import framequery as fq

# assign labels to the generated features
features = fq.execute("""
    SELECT f.*
       ,case when g.type = 'P' then 1 else 0 end as label
    FROM features f
    JOIN game_df g
      on f.game_id = g.game_id
""")
```

We can re-use the logistic regression code from above to build a model that predicts whether an NHL game is a regular or postseason game. The updated code snippet to build a logistic regression model with scikit-learn is shown below. We drop the `game_id` column before fitting the model to avoid training the model on this attribute, which typically results in overfitting.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

# create inputs for sklearn
y = features['label']
X = features.drop(['label', 'game_id'], axis=1).fillna(0)

# train a classifier
lr = LogisticRegression()
model = lr.fit(X, y)

# Results
print("Accuracy: " + str(model.score(X, y)))
print("ROC" + str(roc_auc_score(y,model.predict_proba(X)[:,:1])))
```

The result of this model was an accuracy of 94.7% and an ROC measure of 0.923. While we likely could have created a better performing model by manually specifying how to aggregate play

events into a game summary, we were able to build a model with good accuracy while automating much of this process.

1.8 Conclusion

Building data products is becoming an essential competency for applied data scientists. The Python ecosystem provides useful tools for taking prototype models and scaling them up to production-quality systems. In this chapter, we laid the groundwork for the rest of this book by introducing the data sets, coding tools, cloud environments, and predictive models that we'll use to build scalable model pipelines. We also explored a recent Python library called FeatureTools, which enables automating much of the feature engineering steps in a model pipeline.

In our current setup, we built a simple batch model on a single machine in the cloud. In the next chapter, we'll explore how to share our models with the world, by exposing them as endpoints on the web.

2

Models as Web Endpoints

In order for a machine learning model to be useful, you need a way of sharing the results with other services and applications within your organization. While you can precompute results and save them to a database using a batch pipeline approach, it's often necessary to respond to requests in real-time with up-to-date information. One way of achieving this goal is by setting up a predictive model as a web endpoint that can be invoked from other services. This chapter shows how to set up this functionality for both scikit-learn and Keras models, and introduces Python tools that can help scale up this functionality.

It's good to build experience both hosting and consuming web endpoints when building out model pipelines with Python. In some cases, a predictive model will need to pull data points from other services before making a prediction, such as needing to pull additional attributes about a user's history as input to feature engineering. In this chapter, we'll focus on JSON based services, because it is a popular data format and works well with Python's data types.

A model as an endpoint is a system that provides a prediction in response to a passed in set of parameters. These parameters can be a feature vector, image, or other type of data that is used as input to a predictive model. The endpoint then makes a prediction and returns the results, typically as a JSON payload. The benefits of setting up a model this way are that other systems can use the predictive model, it provides a real-time result, and can be used within a broader data pipeline.

In this chapter, we'll cover calling web services using Python, setting up endpoints, saving models so that they can be used in production environments, hosting scikit-learn and Keras predictive

models, scaling up a service with Gunicorn and Heroku, and building an interactive web application with Plotly Dash.

2.1 Web Services

Before we host a predictive model, we'll use Python to call a web service and to process the result. After showing how to process a web response, we'll set up our own service that echoes the passed in message back to the caller. There's a few different libraries we'll need to install for the examples in this chapter:

```
pip install --user requests
pip install --user flask
pip install --user gunicorn
pip install --user mlflow
pip install --user pillow
pip install --user dash
```

These libraries provide the following functionality:

- **requests:** Provides functions for GET and POST commands.
- **flask:** Enables functions to be exposed as HTTP locations.
- **gunicorn:** A WSGI server that enables hosting Flask apps in production environments.
- **mlflow:** A model library that provides model persistence.
- **pillow:** A fork of the Python Imaging Library.
- **dash:** Enables writing interactive web apps in Python.

Many of the tools for building web services in the Python ecosystem work well with Flask. For example, Gunicorn can be used to host Flask applications at production scale, and the Dash library builds on top of Flask.

To get started with making web requests in Python, we'll use the Cat Facts Heroku app¹. Heroku is a cloud platform that works

¹<https://cat-fact.herokuapp.com/#/>

well for hosting Python applications that we'll explore later in this chapter. The Cat Facts service provides a simple API that provides a JSON response containing interesting tidbits about felines. We can use the `/facts/random` endpoint to retrieve a random fact using the requests library:

```
import requests

result = requests.get("http://cat-fact.herokuapp.com/facts/random")
print(result)
print(result.json())
print(result.json()['text'])
```

This snippet loads the requests library and then uses the `get` function to perform an HTTP get for the passed in URL. The result is a response object that provides a response code and payload if available. In this case, the payload can be processed using the `json` function, which returns the payload as a Python dictionary. The three print statements show the response code, the full payload, and the value for the `text` key in the returned dictionary object. The output for a run of this script is shown below.

```
<Response [200]>

{'used': False, 'source': 'api', 'type': 'cat', 'deleted': False,
 '_id': '591f98c5d1f17a153828aa0b', '__v': 0, 'text':
 'Domestic cats purr both when inhaling and when exhaling.',
 'updatedAt': '2019-05-19T20:22:45.768Z',
 'createdAt': '2018-01-04T01:10:54.673Z'}
```

Domestic cats purr both when inhaling and when exhaling.

2.1.1 Echo Service

Before we set up a complicated environment for hosting a predictive model, we'll start with a simple example. The first service we'll set up is an echo application that returns the passed in mes-

sage parameter as part of the response payload. To implement this functionality, we'll use Flask to build a web service hosted on an EC2 instance. This service can be called on the open web, using the public IP of the EC2 instance. You can also run this service on your local machine, but it won't be accessible over the web. In order to access the function, you'll need to enable access on port 5000, which is covered in Section 1.4.1. The complete code for the echo web service is shown below:

```
import flask
app = flask.Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def predict():
    data = {"success": False}

    # check for passed in parameters
    params = flask.request.json
    if params is None:
        params = flask.request.args

    # if parameters are found, echo the msg parameter
    if "msg" in params.keys():
        data["response"] = params.get("msg")
        data["success"] = True

    return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

The first step is loading the Flask library and creating a Flask object using the name special variable. Next, we define a `predict` function with a Flask annotation that specifies that the function should be hosted at “/” and accessible by HTTP `GET` and `POST` commands. The last step specifies that the application should run using `0.0.0.0` as the host, which enables remote machines to access

the application. By default, the application will run on port 5000, but it's possible to override this setting with the `port` parameter. When running Flask directly, we need to call the `run` function, but we do not want to call the command when running as a module within another application, such as Gunicorn.

The `predict` function returns a JSON response based on the passed in parameters. In Python, you can think of a JSON response as a dictionary, because the `jsonify` function in Flask makes the translation between these data formats seamless. The function first defines a dictionary with the `success` key set to `False`. Next, the function checks if the `request.json` or `request.args` values are set, which indicates that the caller passed in arguments to the function, which we'll cover in the next code snippet. If the user has passed in a `msg` parameter, the `success` key is set to `True` and a `response` key is set to the `msg` parameter in the dictionary. The result is then returned as a JSON payload.

Instead of running this in a Jupyter notebook, we'll save the script as a file called `echo.py`. To launch the Flask application, run `python3 echo.py` on the command line. The result of running this command is shown below:

```
python3 echo.py
* Serving Flask app "echo" (lazy loading)
* Environment: production
  WARNING: This is a development server.
  Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The output indicates that the service is running on port 5000. If you launched the service on your local machine, you can browse to <http://localhost:5000> to call the application, and if using EC2 you'll need to use the public IP, such as <http://52.90.199.190:5000>. The result will be `{"response":null,"success":false}`, which indi-

cates that the service was called but that no message was provided to the echo service.

We can pass parameters to the web service using a few different approaches. The parameters can be appended to the URL, specified using the `params` object when using a GET command, or passed in using the `json` parameter when using a POST command. The snippet below shows how to perform these types of requests. For small sets of parameters, the GET approach works fine, but for larger parameters, such as sending images to a server, the POST approach is preferred.

```
import requests

result = requests.get("http://52.90.199.190:5000/?msg=HelloWorld!")
print(result.json())

result = requests.get("http://52.90.199.190:5000/",
                      params = { 'msg': 'Hello from params' })
print(result.json())

result = requests.post("http://52.90.199.190:5000/",
                      json = { 'msg': 'Hello from data' })
print(result.json())
```

The output of the code snippet is shown below. There are 3 JSON responses showing that the service successfully received the message parameter and echoed the response:

```
{'response': 'HelloWorld!', 'success': True}
{'response': 'Hello from params', 'success': True}
{'response': 'Hello from data', 'success': True}
```

In addition to passing values to a service, it can be useful to pass larger payloads, such as images when hosting deep learning models. One way of achieving this task is by encoding images as strings, which will work with our existing echo service. The code snippet

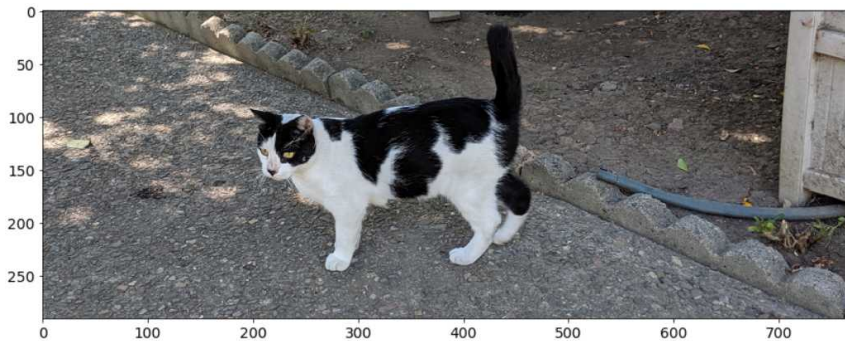


FIGURE 2.1: Passing an image to the echo web service.

below shows how to read in an image and perform base64 encoding on the image before adding it to the request object. The echo service responds with the image payload and we can use the `PIL` library to render the image as a plot.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import io
import base64

image = open("luna.png", "rb").read()
encoded = base64.b64encode(image)
result = requests.get("http://52.90.199.190:5000/",
                      json = {'msg': encoded})
encoded = result.json()['response']
imgData = base64.b64decode(encoded)
plt.imshow( np.array(Image.open(io.BytesIO(imgData))))
```

We can run the script within a Jupyter notebook. The script will load the image and send it to the server, and then render the result as a plot. The output of this script, which uses an image of my in-laws' cat, is shown in Figure 2.1. We won't work much with image data in this book, but I did want to cover how to use more complex objects with web endpoints.

2.2 Model Persistence

To host a model as a web service, we need to provide a model object for the predict function. We can train the model within the web service application, or we can use a pre-trained model. Model persistence is a term used for saving and loading models within a predictive model pipeline. It's common to train models in a separate workflow than the pipeline used to serve the model, such as a Flask application. In this section, we'll save and load both scikit-learn and Keras models, with both direct serialization and the MLFlow library. The goal of saving and loading these models is to make the logistic regression and deep learning models we built in Chapter 1 available as web endpoints.

2.2.1 Scikit-Learn

We'll start with scikit-learn, which we previously used to build a propensity model for identifying which players were most likely to purchase a game. A simple `LogisticRegression` model object can be created using the following script:

```
import pandas as pd
from sklearn.linear_model import LogisticRegression

df = pd.read_csv("https://github.com/bgweber/Twitch/
                 raw/master/Recommendations/games-expand.csv")
x = df.drop(['label'], axis=1)
y = df['label']

model = LogisticRegression()
model.fit(x, y)
```

The default way of saving scikit-learn models is by using pickle, which provides serialization of Python objects. You can save a model using `dump` and load a model using the `load` function, as

shown below. Once you have loaded a model, you can use the prediction functions, such as `predict_proba`.

```
import pickle
pickle.dump(model, open("logit.pkl", 'wb'))

model = pickle.load(open("logit.pkl", 'rb'))
model.predict_proba(x)
```

Pickle is great for simple workflows, but can run into serialization issues when your execution environment is different from your production environment. For example, you might train models on your local machine using Python 3.7 but need to host the models on an EC2 instance running Python 3.6 with different library versions installed.

MLflow is a broad project focused on improving the lifecycle of machine learning projects. The `Models` component of this platform focuses on making models deployable across a diverse range of execution environments. A key goal is to make models more portable, so that your training environment does not need to match your deployment environment. In the current version of MLflow, many of the save and load functions wrap direct serialization calls, but future versions will be focused on using generalized model formats.

We can use MLflow to save a model using `sklearn.save_model` and load a model using `sklearn.load_model`. The script below shows how to perform the same task as the prior code example, but uses MLflow in place of pickle. The file is saved at the `model_path` location, which is a relative path. There's also a commented out command, which needs to be uncommented if the code is executed multiple times. MLflow currently throws an exception if a model is already saved at the current location, and the `rmtee` command can be used to overwrite the existing model.

```
import mlflow
import mlflow.sklearn
```

```
import shutil

model_path = "models/logit_games_v1"
#shutil.rmtree(model_path)
mlflow.sklearn.save_model(model, model_path)

loaded = mlflow.sklearn.load_model(model_path)
loaded.predict_proba(x)
```

2.2.2 Keras

Keras provides built-in functionality for saving and loading deep learning models. We covered building a Keras model for the games data set in Section 1.6.3. The key steps in this process are shown in the following snippet:

```
import tensorflow as tf
import keras
from keras import models, layers

# define the network structure
model = models.Sequential()
model.add(layers.Dense(64,activation='relu',input_shape=(10,)))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

def auc(y_true, y_pred):
    auc = tf.metrics.auc(y_true, y_pred)[1]
    keras.backend.get_session().run(
        tf.local_variables_initializer())
    return auc

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy', metrics=[auc])
```

```
history = model.fit(x, y, epochs=100, batch_size=100,  
                    validation_split = .2, verbose=0)
```

Once we have trained a Keras model, we can use the `save` and `load_model` functions to persist and reload the model using the h5 file format. One additional step here is that we need to pass the custom `auc` function we defined as a metric to the load function in order to reload the model. Once the model is loaded, we can call the prediction functions, such as `evaluate`.

```
from keras.models import load_model  
model.save("games.h5")  
  
model = load_model('games.h5', custom_objects={'auc': auc})  
model.evaluate(x, y, verbose = 0)
```

We can also use MLflow for Keras. The `save_model` and `load_model` functions can be used to persist Keras models. As before, we need to provide the custom-defined `auc` function to load the model.

```
import mlflow.keras  
  
model_path = "models/keras_games_v1"  
mlflow.keras.save_model(model, model_path)  
  
loaded = mlflow.keras.load_model(model_path,  
                                 custom_objects={'auc': auc})  
loaded.evaluate(x, y, verbose = 0)
```

2.3 Model Endpoints

Now that we know how to set up a web service and load pre-trained predictive models, we can set up a web service that provides a

2.3.1 Scikit-Learn

[illegible]

```
data["response"] = str(model.predict_proba(new_x)[0][1])
data["success"] = True

return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

After loading the required libraries, we use `load_model` to load the scikit-learn model object using MLflow. In this setup, the model is loaded only once, and will not be updated unless we relaunch the application. The main change from the echo service is creating the feature vector that we need to pass as input to the model's prediction functions. The `new_row` object creates a dictionary using the passed in parameters. To provide the Pandas row format needed by scikit-learn, we can create a Pandas dataframe based on the dictionary and then transpose the result, which creates a dataframe with a single row. The resulting dataframe is then passed to `predict_proba` to make a propensity prediction for the passed in user. The model output is added to the JSON payload under the `response` key.

Similar to the echo service, we'll need to save the app as a Python file rather than running the code directly in Jupyter. I saved the code as `predict.py` and launched the endpoint by running `python3 predict.py`, which runs the service on port 5000.

To test the service, we can use Python to pass in a record representing an individual user. The dictionary defines the list of games that the user has previously purchased, and the GET command is used to call the service. For the example below, the response key returned a value of 0.3812. If you are running this script in a Jupyter notebook on an EC2 instance, you'll need to enable remote access for the machine on port 5000. Even though the web service and notebook are running on the same machine, we are using the public IP to reference the model endpoint.

```
import requests

new_row = { "G1": 0, "G2": 0, "G3": 0, "G4": 0, "G5": 0,
            "G6": 0, "G7": 0, "G8": 0, "G9": 0, "G10": 1 }

result = requests.get("http://52.90.199.190:5000/", params=new_row)
print(result.json()['response'])
```

2.3.2 Keras

The setup for Keras is similar to scikit-learn, but there are a few additions that need to be made to handle the TensorFlow graph context. We also need to redefine the auc function prior to loading the model using MLflow. The snippet below shows the complete code for a Flask app that serves a Keras model for the game purchases data set.

The main thing to note in this script is the use of the `graph` object. Because Flask uses multiple threads, we need to define the graph used by Keras as a global object, and grab a reference to the graph using the `with` statement when serving requests.

```
import pandas as pd
import mlflow
import mlflow.keras
import flask
import tensorflow as tf
import keras as k

def auc(y_true, y_pred):
    auc = tf.metrics.auc(y_true, y_pred)[1]
    k.backend.get_session().run(
        tf.local_variables_initializer())
    return auc

global graph
```



```

graph = tf.get_default_graph()
model_path = "models/keras_games_v1"
model = mflow.keras.load_model(model_path,
                                custom_objects={'auc': auc})

app = flask.Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def predict():
    data = {"success": False}
    params = flask.request.args

    if "G1" in params.keys():
        new_row = { "G1": params.get("G1"), "G2": params.get("G2"),
                    "G3": params.get("G3"), "G4": params.get("G4"),
                    "G5": params.get("G5"), "G6": params.get("G6"),
                    "G7": params.get("G7"), "G8": params.get("G8"),
                    "G9": params.get("G9"), "G10": params.get("G10") }

        new_x = pd.DataFrame.from_dict(new_row,
                                       orient = "index").transpose()

        with graph.as_default():
            data["response"] = str(model.predict(new_x)[0][0])
            data["success"] = True

    return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')

```

I saved the script as `keras_predict.py` and then launched the Flask app using `python3 keras_predict.py`. The result is a Keras model running as a web service on port 5000. To test the script, we can run the same script from the following section where we tested a scikit-learn model.

2.4 Deploying a Web Endpoint

Flask is great for prototyping models as web services, but it's not intended to be used directly in a production environment. For a proper deployment of a web application, you'll want to use a WSGI Server, which provides scaling, routing, and load balancing. If you're looking to host a web service that needs to handle a large workload, then Gunicorn provides a great solution. If instead you'd like to use a hosted solution, then Heroku provides a platform for hosting web services written in Python. Heroku is useful for hosting a data science portfolio, but is limited in terms of components when building data and model pipelines.

2.4.1 Gunicorn

We can use Gunicorn to provide a WSGI server for our echo Flask application. Using gunicorn helps separate the functionality of an application, which we implemented in Flask, with the deployment of an application. Gunicorn is a lightweight WSGI implementation that works well with Flask apps.

It's straightforward to switch from using Flask directly to using Gunicorn to run the web service. The new command for running the application is shown below. Note that we are passing in a bind parameter to enable remote connections to the service.

```
gunicorn --bind 0.0.0.0 echo:app
```

The result on the command line is shown below. The main difference from before is that we now interface with the service on port 8000 rather than on port 5000. If you want to test out the service, you'll need to enable remote access on port 8000.

```
gunicorn --bind 0.0.0.0 echo:app
[INFO] Starting gunicorn 19.9.0
[INFO] Listening at: http://0.0.0.0:8000 (9509)
```

```
[INFO] Using worker: sync
[INFO] Booting worker with pid: 9512
```

To test the service using Python, we can run the following snippet. You'll need to make sure that access to port 8000 is enabled, as discussed in Section 1.4.1.

```
result = requests.get("http://52.90.199.190:8000/",
                      params = { 'msg': 'Hello from Gunicorn' })
print(result.json())
```

The result is a JSON response with the passed in message. The main distinction from our prior setup is that we are now using Gunicorn, which can use multiple threads to handle load balancing, and can perform additional server configuration that is not available when using only Flask. Configuring Gunicorn to serve production workloads is outside the scope of this book, because it is a hosted solution where a team needs to manage DevOps of the system. Instead, we'll focus on managed solutions, including AWS Lambda and Cloud Functions in Chapter 3, where minimal overhead is needed to keep systems operational.

2.4.2 Heroku

Now that we have a Gunicorn application, we can host it in the cloud using Heroku. Python is one of the core languages supported by this cloud environment. The great thing about using Heroku is that you can host apps for free, which is great for showcasing data science projects. The first step is to set up an account on the web site: <https://www.heroku.com/>

Next, we'll set up the command line tools for Heroku, by running the commands shown below. There can be some complications when setting up Heroku on an AMI EC2 instance, but downloading and unzipping the binaries directly works around these problems. The steps shown below download a release, extract it, and install an additional dependency. The last step outputs the version of

Heroku installed. I got the following output: `heroku/7.29.0 linux-x64 node-v11.14.0`.

```
wget https://cli-assets.heroku.com/heroku-linux-x64.tar.gz
unzip heroku-linux-x64.tar.gz
tar xf heroku-linux-x64.tar
sudo yum -y install glibc.i686
/home/ec2-user/heroku/bin/heroku --version
```

Once Heroku is installed, we need to set up a project for where we will deploy projects. We can use the CLI to create a new Heroku project by running the following commands:

```
/home/ec2-user/heroku/bin/heroku login
/home/ec2-user/heroku/bin/heroku create
```

This will create a unique app name, such as `obscure-coast-69593`. It's good to test the setup locally before deploying to production. In order to test the setup, you'll need to install the `django` and `django-heroku` packages. Heroku has some dependencies on Postgres, which is why additional `install` and `easy_install` commands are included when installing these libraries.

```
pip install --user django
sudo yum install gcc python-setuptools postgresql-devel
sudo easy_install psycopg2
pip install --user django-heroku
```

To get started with building a Heroku application, we'll first download the sample application from GitHub and then modify the project to include our echo application.

```
sudo yum install git
git clone https://github.com/heroku/python-getting-started.git
cd python-getting-started
```

Next, we'll make our changes to the project. We copy our `echo.py` file into the directory, add Flask to the list of dependencies in the `requirements.txt` file, override the command to run in the `Procfile`, and then call `heroku local` to test the configuration locally.

```
cp ../echo.py echo.py
echo 'flask' >> requirements.txt
echo "web: gunicorn echo:app" > Procfile
/home/ec2-user/heroku/bin/heroku local
```

You should see a result that looks like this:

```
/home/ec2-user/heroku/bin/heroku local
[OKAY] Loaded ENV .env File as KEY=VALUE Format
[INFO] Starting gunicorn 19.9.0
[INFO] Listening at: http://0.0.0.0:5000 (10485)
[INFO] Using worker: sync
[INFO] Booting worker with pid: 10488
```

As before, we can test the endpoint using a browser or a Python call, as shown below. In the Heroku local test configuration, port 5000 is used by default.

```
result = requests.get("http://localhost:5000/",
                      params = { 'msg': 'Hello from Heroku Local'})
print(result.json())
```

The final step is to deploy the service to production. The git commands are used to push the results to Heroku, which automatically releases a new version of the application. The last command tells Heroku to scale up to a single worker, which is free.

```
git add echo.py
git commit .
git push heroku master
/home/ec2-user/heroku/bin/heroku ps:scale web=1
```

After these steps run, there should be a message that the application has been deployed to Heroku. Now we can call the endpoint, which has a proper URL, is secured, and can be used to publicly share data science projects.

```
result = requests.get("https://obscure-coast-69593.herokuapp.com",  
                      params = { 'msg': 'Hello from Heroku Prod' })  
print(result.json())
```

There's many languages and tools supported by Heroku, and it's useful for hosting small-scale data science projects.

2.5 Interactive Web Services

While the standard deployment of a model as a web service is an API that you can call programmatically, it's often useful to expose models as interactive web applications. For example, we might want to build an application where there is a UI for specifying different inputs to a model, and the UI reacts to changes made by the user. While Flask can be used to build web pages that react to user input, there are libraries built on top of Flask that provide higher-level abstractions for building web applications with the Python language.

2.5.1 Dash

Dash is a Python library written by the Plotly team that enables building interactive web applications with Python. You specify an application layout and a set of callbacks that respond to user input. If you've used Shiny in the past, Dash shares many similarities, but is built on Python rather than R. With Dash, you can create simple applications as we'll show here, or complex dashboards that interact with machine learning models.

We'll create a simple Dash application that provides a UI for interacting with a model. The application layout will contain three

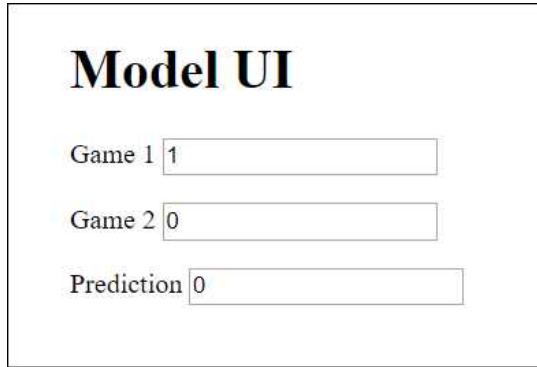
text boxes, where two of these are for user inputs and the third one shows the output of the model. We'll create a file called `dash_app.py` and start by specifying the libraries to import.

```
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output
import pandas as pd
import mlflow.sklearn
```

Next, we'll define the layout of our application. We create a Dash object and then set the `layout` field to include a title and three text boxes with labels. We'll include only 2 of the 10 games from the games data set, to keep the sample short. The last step in the script launches the web service and enables connections from remote machines.

```
app = dash.Dash(__name__)

app.layout = html.Div(children=[
    html.H1(children='Model UI'),
    html.P([
        html.Label('Game 1 '),
        dcc.Input(value='1', type='text', id='g1'),
    ]),
    html.Div([
        html.Label('Game 2 '),
        dcc.Input(value='0', type='text', id='g2'),
    ]),
    html.P([
        html.Label('Prediction '),
        dcc.Input(value='0', type='text', id='pred')
    ]),
])
```



Model UI

Game 1

Game 2

Prediction

FIGURE 2.2: The initial Dash application.

```
if __name__ == '__main__':  
    app.run_server(host='0.0.0.0')
```

Before writing the callbacks, we can test out the layout of the application by running `python3 dash_app.py`, which will run on port 8050 by default. You can browse to your public IP on port 8050 to see the resulting application. The initial application layout is shown in Figure 2.2. Before any callbacks are added, the result of the Prediction text box will always be 0.

The next step is to add a callback to the application so that the Prediction text box is updated whenever the user changes one of the Game 1 or Game 2 values. To perform this task, we define a callback shown in the snippet below. The callback is defined after the application layout, but before the `run_server` command. We also load the logistic regression model for the games data set using MLflow. The callback uses an annotation to define the inputs to the function, the output, and any additional state that needs to be provided. The way that the annotation is defined here, the function will be called whenever the value of Game 1 or Game 2 is modified by the user, and the value returned by this function will be set as the value of the Prediction text box.


```
model_path = "models/logit_games_v1"
model = mlflow.sklearn.load_model(model_path)

@app.callback(
    Output(component_id='pred', component_property='value'),
    [Input(component_id='g1', component_property='value'),
     Input(component_id='g2', component_property='value')]
)
def update_prediction(game1, game2):

    new_row = { "G1": float(game1),
                "G2": float(game2),
                "G3": 0, "G4": 0,
                "G5": 0, "G6": 0,
                "G7": 0, "G8": 0,
                "G9": 0, "G10":0 }

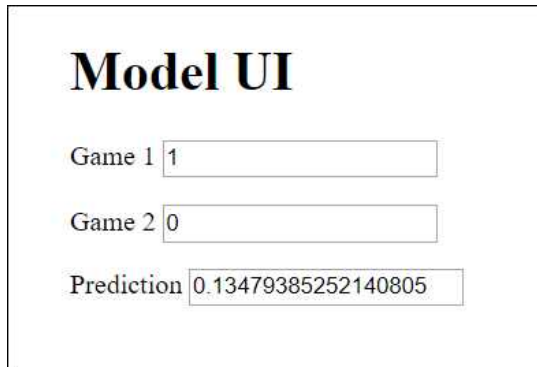
    new_x = pd.DataFrame.from_dict(new_row,
                                   orient = "index").transpose()

    return str(model.predict_proba(new_x)[0][1])
```

The function takes the two values provided by the user, and creates a Pandas dataframe. As before, we transpose the dataframe to provide a single row that we'll pass as input to the loaded model. The value predicted by the model is then returned and set as the value of the Prediction text box.

The updated application with the callback function included is shown in Figure 2.3. The prediction value now dynamically changes in response to changes in the other text fields, and provides a way of introspecting the model.

Dash is great for building web applications, because it eliminates the need to write JavaScript code. It's also possible to stylize Dash application using CSS to add some polish to your tools.



Model UI

Game 1

Game 2

Prediction

FIGURE 2.3: The resulting model prediction.

2.6 Conclusion

The Python ecosystem has a great suite of tools for building web applications. Using only Python, you can write scalable APIs deployed to the open web or custom UI applications that interact with backend Python code. This chapter focused on Flask, which can be extended with other libraries and hosted in a wide range of environments. One of the important concepts we touched on in this chapter is model persistence, which will be useful in other contexts when building scalable model pipelines. We also deployed a simple application to Heroku, which is a separate cloud platform from AWS and GCP.

This chapter is only an introduction to the many different web tools within the Python ecosystem, and the topic of scaling these types of tools is outside the scope of this book. Instead, we'll focus on managed solutions for models on the web, which significantly reduces the DevOps overhead of deploying models as web services. The next chapter will cover two systems for serverless functions in managed environments.

3

Models as Serverless Functions

Serverless technologies enable developers to write and deploy code without needing to worry about provisioning and maintaining servers. One of the most common uses of this technology is serverless functions, which makes it much easier to author code that can scale to match variable workloads. With serverless function environments, you write a function that the runtime supports, specify a list of dependencies, and then deploy the function to production. The cloud platform is responsible for provisioning servers, scaling up more machines to match demand, managing load balancers, and handling versioning. Since we've already explored hosting models as web endpoints, serverless functions are an excellent tool to utilize when you want to rapidly move from prototype to production for your predictive models.

Serverless functions were first introduced on AWS in 2015 and GCP in 2016. Both of these systems provide a variety of triggers that can invoke functions, and a number of outputs that the functions can trigger in response. While it's possible to use serverless functions to avoid writing complex code for glueing different components together in a cloud platform, we'll explore a much narrower use case in this chapter. We'll write serverless functions that are triggered by an HTTP request, calculate a propensity score for the passed in feature vector, and return the prediction as JSON. For this specific use case, GCP's Cloud Functions are much easier to get up and running, but we'll explore both AWS and GCP solutions.

In this chapter, we'll introduce the concept of managed services, where the cloud platform is responsible for provisioning servers. Next, we'll cover hosting sklearn and Keras models with Cloud

Functions. To conclude, we'll show how to achieve the same result for sklearn models with Lambda functions in AWS. We'll also touch on model updates and access control.

3.1 Managed Services

Since 2015, there's been a movement in cloud computing to transition developers away from manually provisioning servers to using managed services that abstract away the concept of servers. The main benefit of this new paradigm is that developers can write code in a staging environment and then push code to production with minimal concerns about operational overhead, and the infrastructure required to match the required workload can be automatically scaled as needed. This enables both engineers and data scientists to be more active in DevOps, because much of the operational concerns of the infrastructure are managed by the cloud provider.

Manually provisioning servers, where you `ssh` into the machines to set up libraries and code, is often referred to as *hosted* deployments, versus *managed* solutions where the cloud platform is responsible for abstracting away this concern from the user. In this book, we'll cover examples in both of these categories. Here are some of the different use cases we'll cover:

- **Web Endpoints:** Single EC2 instance (hosted) vs AWS Lambda (managed).
- **Docker:** Single EC2 instance (hosted) vs ECS (managed).
- **Messaging:** Kafka (hosted) vs PubSub (managed).

This chapter will walk through the first use case, migrating web endpoints from a single machine to an elastic environment. We'll also work through examples that thread this distinction, such as deploying Spark environments with specific machine configurations and manual cluster management.

Serverless technologies and managed services are a powerful tool for data scientists, because they enable a single developer to build

data pipelines that can scale to massive workloads. It's a powerful tool for data scientists to wield, but there are a few trade-offs to consider when using managed services. Here are some of the main issues to consider when deciding between hosted and managed solutions:

- **Iteration:** Are you rapidly prototyping on a product or iterating on a system in production?
- **Latency:** Is a multi-second latency acceptable for your SLAs?
- **Scale:** Can your system scale to match peak workload demands?
- **Cost:** Are you willing to pay more for serverless cloud costs?

At a startup, serverless technologies are great because you have low-volume traffic and have the ability to quickly iterate and try out new architectures. At a certain scale, the dynamics change and the cost of using serverless technologies may be less appealing when you already have in-house expertise for provisioning cloud services. In my past projects, the top issue that was a concern was latency, because it can impact customer experiences. In chapter 8, we'll touch on this topic, because managed solutions often do not scale well to large streaming workloads.

Even if your organization does not use managed services in daily operations, it's a useful skill set to get hands on with as a data scientist, because it means that you can separate model training from model deployment issues. One of the themes in this book is that models do not need to be complex, but it can be complex to deploy models. Serverless functions are a great approach for demonstrating the ability to serve models at scale, and we'll walk through two cloud platforms that provide this capability.

3.2 Cloud Functions (GCP)

Google Cloud Platform provides an environment for serverless functions called Cloud Functions. The general concept with this tool is that you can write code targeted for Flask, but leverage the managed services in GCP to provide elastic computing for your

Python code. GCP is a great environment to get started with serverless functions, because it closely matches standard Python development ecosystems, where you specify a requirements file and application code.

We'll build scalable endpoints that serve both `sklearn` and `Keras` models with Cloud Functions. There are a few issues to be aware of when writing functions in this environment:

- **Storage:** Cloud Functions run in a read-only environment, but you can write to the `/tmp` directory.
- **Tabs:** Spaces versus tabs can cause issues in Cloud Functions, and if you are working in the web editor versus familiar tools like Sublime Text, these can be difficult to spot.
- **sklearn:** When using a requirements file, it's important to differentiate between `sklearn` and `scikit-learn` based on your imports. We'll use `sklearn` in this chapter.

Cloud platforms are always changing, so the specific steps outlined in this chapter may change based on the evolution of these platforms, but the general approach for deploying functions should apply throughout these updates. As always, the approach I advocate for is starting with a simple example, and then scaling to more complex solutions as needed. In this section, we'll first build an echo service and then explore `sklearn` and `Keras` models.

3.2.1 Echo Service

GCP provides a web interface for authoring Cloud Functions. This UI provides options for setting up the triggers for a function, specifying the requirements file for a Python function, and authoring the implementation of the Flask function that serves the request. To start, we'll set up a simple echo service that reads in a parameter from an HTTP request and returns the passed in parameter as the result.

In GCP, you can directly set up a Cloud Function as an HTTP endpoint without needing to configure additional triggers. To get started with setting up an echo service, perform the following actions in the GCP console:

FIGURE 3.1: Creating a Cloud Function.

1. Search for “Cloud Function”
2. Click on “Create Function”
3. Select “HTTP” as the trigger
4. Select “Allow unauthenticated invocations”
5. Select “Inline Editor” for source code
6. Select Python 3.7 as the runtime

An example of this process is shown in Figure 3.1. After performing these steps, the UI will provide tabs for the `main.py` and `requirements.txt` files. The `requirements` file is where we will specify libraries, such as `flask >= 1.1.1`, and the `main` file is where we’ll implement our function behavior.

We’ll start by creating a simple echo service that parses out the `msg` parameter from the passed in request and returns this parameter as a JSON response. In order to use the `jsonify` function we need

to include the `flask` library in the requirements file. The `requirements.txt` file and `main.py` files for the simple echo service are shown in the snippet below. The echo function here is similar to the echo service we coded in Section 2.1.1, the main distinction here is that we are no longer using annotations to specify the endpoints and allowed methods. Instead, these settings are now being specified using the Cloud Functions UI.

```
# requirements.txt
flask

#main.py
def echo(request):
    from flask import jsonify

    data = {"success": False}
    params = request.get_json()

    if "msg" in params:
        data["response"] = str(params['msg'])
        data["success"] = True

    return jsonify(data)
```

We can deploy the function to production by performing the following steps:

1. Update “Function to execute” to “echo”
2. Click “Create” to deploy

Once the function has been deployed, you can click on the “Testing” tab to check if the deployment of the function worked as intended. You can specify a JSON object to pass to the function, and invoke the function by clicking “Test the function”, as shown in Figure 3.2. The result of running this test case is the JSON object returned in the output dialog, which shows that invoking the echo function worked correctly.

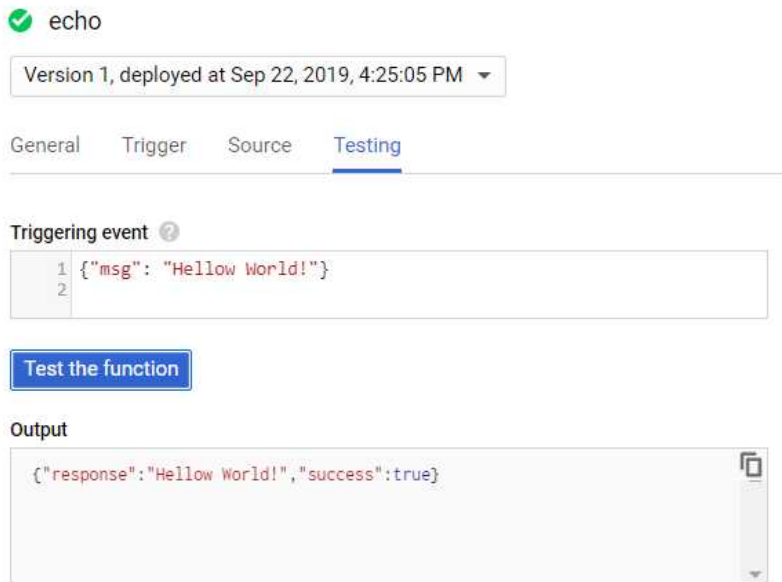


FIGURE 3.2: Testing a Cloud Function.

Now that the function is deployed and we enabled unauthenticated access to the function, we can call the function over the web using Python. To get the URL of the function, click on the “trigger” tab. We can use the `requests` library to pass a JSON object to the serverless function, as shown in the snippet below.

```
import requests

result = requests.post(
    "https://us-central1-gameanalytics.cloudfunctions.net/echo"
    , json = { 'msg': 'Hello from Cloud Function' })
print(result.json())
```

The result of running this script is that a JSON payload is returned from the serverless function. The output from the call is the JSON shown below.

```
{  
    'response': 'Hello from Cloud Function',  
    'success': True  
}
```

We now have a serverless function that provides an echo service. In order to serve a model using Cloud Functions, we'll need to persist the model specification somewhere that the serverless function can access. To accomplish this, we'll use Cloud Storage to store the model in a distributed storage layer.

3.2.2 Cloud Storage (GCS)

GCP provides an elastic storage layer called Google Cloud Storage (GCS) that can be used for distributed file storage and can also scale to other uses such as data lakes. In this section, we'll explore the first use case of utilizing this service to store and retrieve files for use in a serverless function. GCS is similar to AWS's offering called S3, which is leveraged extensively in the gaming industry to build data platforms.

While GCP does provide a UI for interacting with GCS, we'll explore the command line interface in this section, since this approach is useful for building automated workflows. GCP requires authentication for interacting with this service, please revisit section 1.5.1 if you have not yet set up a JSON credentials file. In order to interact with Cloud Storage using Python, we'll also need to install the GCS library, using the command shown below:

```
pip install --user google-cloud-storage  
export GOOGLE_APPLICATION_CREDENTIALS=/home/ec2-user/dsdemo.json
```

Now that we have the prerequisite libraries installed and credentials set up, we can interact with GCS programmatically using Python. Before we can store a file, we need to set up a bucket on GCS. A bucket is a prefix assigned to all files stored on GCS, and each bucket name must be globally unique. We'll create a bucket

name called `dsp_model_store` where we'll store model objects. The script below shows how to create a new bucket using the `create_bucket` function and then iterate through all of the available buckets using the `list_buckets` function. You'll need to change the `bucket_name` variable to something unique before running this script.

```
from google.cloud import storage

bucket_name = "dsp_model_store"

storage_client = storage.Client()
storage_client.create_bucket(bucket_name)

for bucket in storage_client.list_buckets():
    print(bucket.name)
```

After running this code, the output of the script should be a single bucket, with the name assigned to the `bucket_name` variable. We now have a path on GCS that we can use for saving files: `gs://dsp_model_storage`.

We'll reuse the model we trained in Section 2.2.1 to deploy a logistic regression model with Cloud Functions. To save the file to GCS, we need to assign a path to the destination, shown by the `bucket.blob` command below and select a local file to upload, which is passed to the `upload` function.

```
from google.cloud import storage

bucket_name = "dsp_model_store"
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

blob = bucket.blob("serverless/logit/v1")
blob.upload_from_filename("logit.pkl")
```

After running this script, the local file `logit.pkl` will now be available on GCS at the following location:

```
gs://dsp_model_storage/serverless/logit/v1/logit.pkl
```

While it's possible to use URIs such as this directly to access files, as we'll explore with Spark in Chapter 6, in this section we'll retrieve the file using the bucket name and blob path. The code snippet below shows how to download the model file from GCS to local storage. We download the model file to the local path of `local_logit.pkl` and then load the model by calling `pickle.load` with this path.

```
import pickle
from google.cloud import storage

bucket_name = "dsp_model_store"
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

blob = bucket.blob("serverless/logit/v1")
blob.download_to_filename("local_logit.pkl")
model = pickle.load(open("local_logit.pkl", 'rb'))
model
```

We can now programmatically store model files to GCS using Python and also retrieve them, enabling us to load model files in Cloud Functions. We'll combine this with the Flask examples from the previous chapter to serve sklearn and Keras models as Cloud Functions.

3.2.3 Model Function

We can now set up a Cloud Function that serves logistic regression model predictions over the web. We'll build on the Flask example that we explored in Section 2.3.1 and make a few modifications for the service to run on GCP. The first step is to specify the required Python libraries that we'll need to serve requests in the `requirements.txt` file, as shown below. We'll also need Pandas to set up a dataframe for making the prediction, sklearn for applying

the model, and cloud storage for retrieving the model object from GCS.

```
google-cloud-storage
sklearn
pandas
flask
```

The next step is to implement our model function in the `main.py` file. A small change from before is that the `params` object is now fetched using `request.get_json()` rather than `flask.request.args`. The main change is that we are now downloading the model file from GCS rather than retrieving the file directly from local storage, because local files are not available when writing Cloud Functions with the UI tool. An additional change from the prior function is that we are now reloading the model for every request, rather than loading the model file once at startup. In a later code snippet, we'll show how to use global objects to cache the loaded model.

```
def pred(request):
    from google.cloud import storage
    import pickle as pk
    import sklearn
    import pandas as pd
    from flask import jsonify

    data = {"success": False}
    params = request.get_json()

    if "G1" in params:

        new_row = { "G1": params.get("G1"), "G2": params.get("G2"),
                    "G3": params.get("G3"), "G4": params.get("G4"),
                    "G5": params.get("G5"), "G6": params.get("G6"),
                    "G7": params.get("G7"), "G8": params.get("G8"),
                    "G9": params.get("G9"), "G10": params.get("G10") }
```

```

new_x = pd.DataFrame.from_dict(new_row,
                               orient = "index").transpose()

# set up access to the GCS bucket
bucket_name = "dsp_model_store"
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

# download and load the model
blob = bucket.blob("serverless/logit/v1")
blob.download_to_filename("/tmp/local_logit.pkl")
model = pk.load(open("/tmp/local_logit.pkl", 'rb'))

data["response"] = str(model.predict_proba(new_x)[0][1])
data["success"] = True

return jsonify(data)

```

One note in the code snippet above is that the `/tmp` directory is used to store the downloaded model file. In Cloud Functions, you are unable to write to the local disk, with the exception of this directory. Generally it's best to read objects directly into memory rather than pulling objects to local storage, but the Python library for reading objects from GCS currently requires this approach.

For this function, we created a new Cloud Function named `pred`, set the function to execute to `pred`, and deployed the function to production. We can now call the function from Python, using the same approach from 2.3.1 with a URL that now points to the Cloud Function, as shown below:

```

import requests

result = requests.post(
    "https://us-central1-gameanalytics.cloudfunctions.net/pred"

```

```
,json = { 'G1':'1', 'G2':'0', 'G3':'0', 'G4':'0', 'G5':'0'
          , 'G6':'0', 'G7':'0', 'G8':'0', 'G9':'0', 'G10':'0'})
print(result.json())
```

The result of the Python web request to the function is a JSON response with a response value and model prediction, shown below:

```
{
  'response': '0.06745113592634559',
  'success': True
}
```

In order to improve the performance of the function, so that it takes milliseconds to respond rather than seconds, we'll need to cache the model object between runs. It's best to avoid defining variables outside of the scope of the function, because the server hosting the function may be terminated due to inactivity. Global variables are an execution to this rule, when used for caching objects between function invocations. This code snippet below shows how a global model object can be defined within the scope of the `pred` function to provide a persistent object across calls. During the first function invocation, the model file will be retrieved from GCS and loaded via pickle. During following runs, the model object will already be loaded into memory, providing a much faster response time.

```
model = None

def pred(request):
    global model

    if not model:

        # download model from GCS
        model = pk.load(open("/tmp/local_logit.pkl", 'rb'))
```

```
# apply model

return jsonify(data)
```

Caching objects is important for authoring responsive models that lazily load objects as needed. It's also useful for more complex models, such as Keras which requires persisting a TensorFlow graph between invocations.

3.2.4 Keras Model

Since Cloud Functions provide a requirements file that can be used to add additional dependencies to a function, it's also possible to serve Keras models with this approach. We'll be able to reuse most of the code from the past section, and we'll also use the Keras and Flask approach introduced in Section 2.3.2. Given the size of the Keras libraries and dependencies, we'll need to upgrade the memory available for the Function from 256 MB to 1GB. We also need to update the requirements file to include Keras:

```
google-cloud-storage
tensorflow
keras
pandas
flask
```

The full implementation for the Keras model as a Cloud Function is shown in the code snippet below. In order to make sure that the TensorFlow graph used to load the model is available for future invocations of the model, we use global variables to cache both the model and graph objects. To load the Keras model, we need to redefine the `auc` function that was used during model training, which we include within the scope of the `predict` function. We reuse the same approach from the prior section to download the model file from GCS, but now use `load_model` from Keras to read the model file into memory from the temporary disk location. The

result is a Keras predictive model that lazily fetches the model file and can scale to meet variable workloads as a serverless function.

```
model = None
graph = None

def predict(request):
    global model
    global graph

    from google.cloud import storage
    import pandas as pd
    import flask
    import tensorflow as tf
    import keras as k
    from keras.models import load_model
    from flask import jsonify

    def auc(y_true, y_pred):
        auc = tf.metrics.auc(y_true, y_pred)[1]
        k.backend.get_session().run(
            tf.local_variables_initializer())
        return auc

    data = {"success": False}
    params = request.get_json()

    # download model if not cached
    if not model:
        graph = tf.get_default_graph()

        bucket_name = "dsp_model_store_1"
        storage_client = storage.Client()
        bucket = storage_client.get_bucket(bucket_name)

        blob = bucket.blob("serverless/keras/v1")
```

```

blob.download_to_filename("/tmp/games.h5")
model = load_model('/tmp/games.h5',
                    custom_objects={'auc': auc})

# apply the model
if "G1" in params:
    new_row = { "G1": params.get("G1"), "G2": params.get("G2"),
                "G3": params.get("G3"), "G4": params.get("G4"),
                "G5": params.get("G5"), "G6": params.get("G6"),
                "G7": params.get("G7"), "G8": params.get("G8"),
                "G9": params.get("G9"), "G10": params.get("G10")}

    new_x = pd.DataFrame.from_dict(new_row,
                                   orient = "index").transpose()

    with graph.as_default():
        data["response"] = str(model.predict_proba(new_x)[0][0])
        data["success"] = True

return jsonify(data)

```

To test the deployed model, we can reuse the Python web request script from the prior section and replace `pred` with `predict` in the request URL. We have now deployed a deep learning model to production.

3.2.5 Access Control

The Cloud Functions we introduced in this chapter are open to the web, which means that anyone can access them and potentially abuse the endpoints. In general, it's best not to enable unauthenticated access and instead lock down the function so that only authenticated users and services can access them. This recommendation also applies to the Flask apps that we deployed in the last chapter, where it's a best practice to restrict access to services that can reach the endpoint using AWS private IPs.

There are a few different approaches for locking down Cloud Functions to ensure that only authenticated users have access to the functions. The easiest approach is to disable “Allow unauthenticated invocations” in the function setup to prevent hosting the function on the open web. To use the function, you’ll need to set up IAM roles and credentials for the function. This process involves a number of steps and may change over time as GCP evolves. Instead of walking through this process, it’s best to refer to the GCP documentation¹.

Another approach for setting up functions that enforce authentication is by using other services within GCP. We’ll explore this approach in Chapter 8, which introduces GCP’s PubSub system for producing and consuming messages within GCP’s ecosystem.

3.2.6 Model Refreshes

We’ve deployed sklearn and Keras models to production using Cloud Functions, but the current implementations of these functions use static model files that will not change over time. It’s usually necessary to make changes to models over time to ensure that the accuracy of the models do not drift too far from expected performance. There’s a few different approaches that we can take to update the model specification that a Cloud Function is using:

1. **Redeploy:** Overwriting the model file on GCS and redeploying the function will result in the function loading the updated file.
2. **Timeout:** We can add a timeout to the function, where the model is re-downloaded after a certain threshold of time passes, such as 30 minutes.
3. **New Function:** We can deploy a new function, such as `pred_v2` and update the URL used by systems calling the service, or use a load balancer to automate this process.
4. **Model Trigger:** We can add additional triggers to the function to force the function to manually reload the model.

¹<https://cloud.google.com/functions/docs/securing/authenticating>

While the first approach is the easiest to implement and can work well for small-scale deployments, the third approach, where a load balancer is used to direct calls to the newest function available is probably the most robust approach for production systems. A best practice is to add logging to your function, in order to track predictions over time so that you can log the performance of the model and identify potential drift.

3.3 Lambda Functions (AWS)

AWS also provides an ecosystem for serverless functions called Lambda. AWS Lambda is useful for glueing different components within an AWS deployment together, since it supports a rich set of triggers for function inputs and outputs. While Lambda does provide a powerful tool for building data pipelines, the current Python development environment is a bit clunkier than GCP.

In this section we'll walk through setting up an echo service and an sklearn model endpoint with Lambda. We won't cover Keras, because the size of the library causes problems when deploying a function with AWS. Unlike the past section where we used a UI to define functions, we'll use command line tools for providing our function definition to Lambda.

3.3.1 Echo Function

For a simple function, you can use the inline code editor that Lambda provides for authoring functions. You can create a new function by performing the following steps in the AWS console:

1. Under "Find Services", select "Lambda"
2. Select "Create Function"
3. Use "Author from scratch"
4. Assign a name (e.g. echo)
5. Select a Python runtime
6. Click "Create Function"

After running these steps, Lambda will generate a file called `lambda_function.py`. The file defines a function called `lambda_handler` which we'll use to implement the echo service. We'll make a small modification to the file, as shown below, which echoes the `msg` parameter as the body of the response object.

```
def lambda_handler(event, context):  
  
    return {  
        'statusCode': 200,  
        'body': event['msg']  
    }
```

Click “Save” to deploy the function and then “Test” to test the file. If you use the default test parameters, then an error will be returned when running the function, because no `msg` key is available in the event object. Click on “Configure test event”, and define use the following configuration:

```
{  
    "msg": "Hello from Lambda!"  
}
```

After clicking on “Test”, you should see the execution results. The response should be the echoed message with a status code of 200 returned. There's also details about how long the function took to execute (25.8ms), the billing duration (100ms), and the maximum memory used (56 MB).

We have now a simple function running on AWS Lambda. For this function to be exposed to external systems, we'll need to set up an API Gateway, which is covered in Section 3.3.3. This function will scale up to meet demand if needed, and requires no server monitoring once deployed. To setup a function that deploys a model, we'll need to use a different workflow for authoring and publishing the function, because AWS Lambda does not currently support a `requirements.txt` file for defining dependencies when writing func-

tions with the inline code editor. To store the model file that we want to serve with a Lambda function, we'll use S3 as a storage layer for model artifacts.

3.3.2 Simple Storage Service (S3)

AWS provides a highly-performant storage layer called S3, which can be used to host individual files for web sites, store large files for data processing, and even host thousands or millions of files for building data lakes. For now, our use case will be storing an individual `zip` file, which we'll use to deploy new Lambda functions. However, there are many broader use cases and many companies use S3 as their initial endpoint for data ingestion in data platforms.

In order to use S3 to store our function to deploy, we'll need to set up a new S3 bucket, define a policy for accessing the bucket, and configure credentials for setting up command line access to S3. Buckets on S3 are analogous to GCS buckets in GCP.

To set up a bucket, browse to the AWS console and select "S3" under find services. Next, select "Create Bucket" to set up a location for storing files on S3. Create a unique name for the S3 bucket, as shown in Figure 3.3, and click "Next" and then "Create Bucket" to finalize setting up the bucket.

We now have a location to store objects on S3, but we still need to set up a user before we can use the command line tools to write and read from the bucket. Browse to the AWS console and select "IAM" under "Find Services". Next, click "Users" and then "Add user" to set up a new user. Create a user name, and select "Programmatic access" as shown in Figure 3.4.

The next step is to provide the user with full access to S3. Use the attach existing policies option and search for S3 policies in order to find and select the `AmazonS3FullAccess` policy, as shown in Figure 3.5. Click "Next" to continue the process until a new user is defined. At the end of this process, a set of credentials will be displayed, including an access key ID and secret access key. Store these values in a safe location.

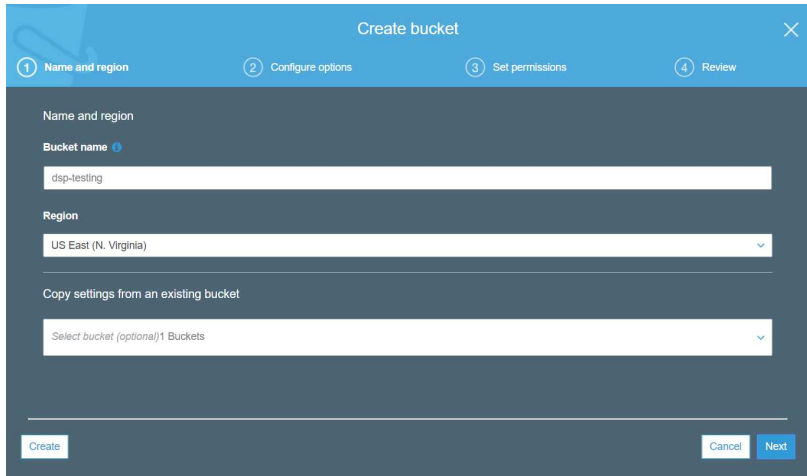


FIGURE 3.3: Creating an S3 bucket on AWS.

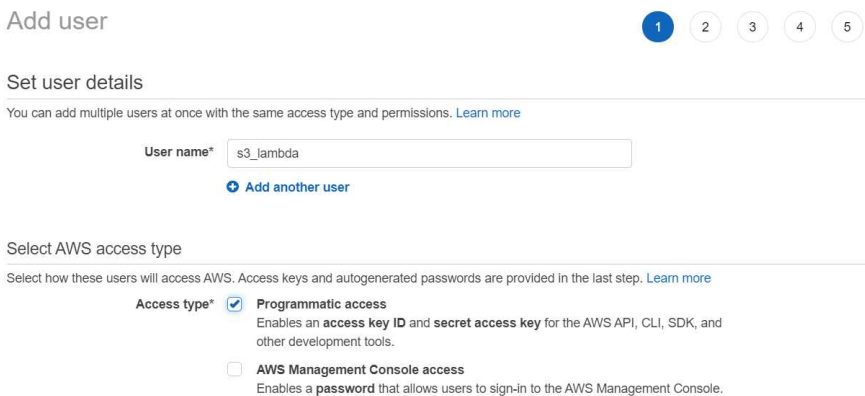


FIGURE 3.4: Setting up a user with S3 access.

The last step needed for setting up command line access to S3 is running the `aws configure` command from your EC2 instance. You'll be asked to provide the access and secret keys from the user we just set up. In order to test that the credentials are properly configured, you can run the following commands:

```
aws configure
aws s3 ls
```

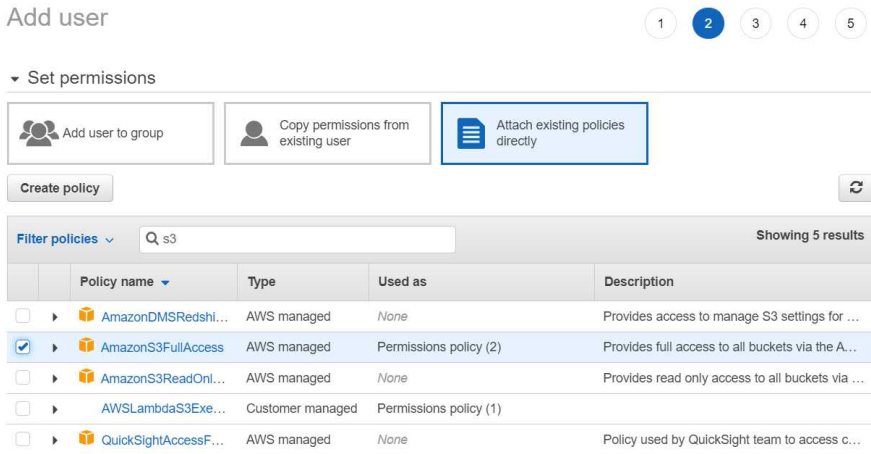


FIGURE 3.5: Selecting a policy for full S3 access.

The results should include the name of the S3 bucket we set up at the beginning of this section. Now that we have an S3 bucket set up with command line access, we can begin writing Lambda functions that use additional libraries such as Pandas and sklearn.

3.3.3 Model Function

In order to author a Lambda function that uses libraries outside of the base Python distribution, you'll need to set up a local environment that defines the function and includes all of the dependencies. Once your function is defined, you can upload the function by creating a zip file of the local environment, uploading the resulting file to S3, and configuring a Lambda function from the file uploaded to S3.

The first step in this process is to create a directory with all of the dependencies installed locally. While it's possible to perform this process on a local machine, I used an EC2 instance to provide a clean Python environment. The next step is to install the libraries needed for the function, which are Pandas and sklearn. These libraries are already installed on the EC2 instance, but need to be reinstalled in the current directory in order to be included in the

zip file that we'll upload to S3. To accomplish this, we can append `-t .` to the end of the `pip` command in order to install the libraries into the current directory. The last steps to run on the command line are copying our logistic regression model into the current directory, and creating a new file that will implement the Lambda function.

```
mkdir lambda
cd lambda
pip install pandas -t .
pip install sklearn -t .
cp ../logit.pkl logit.pkl
vi logit.py
```

The full source code for the Lambda function that serves our logistic regression model is shown in the code snippet below. The structure of the file should look familiar, we first globally define a model object and then implement a function that services model requests. This function first parses the response to extract the inputs to the model, and then calls `predict_proba` on the resulting dataframe to get a model prediction. The result is then returned as a dictionary object containing a `body` key. It's important to define the function response within the `body` key, otherwise Lambda will throw an exception when invoking the function over the web.

```
from sklearn.externals import joblib
import pandas as pd
import json
model = joblib.load('logit.pkl')

def lambda_handler(event, context):

    # read in the request body as the event dict
    if "body" in event:
        event = event["body"]
```

```

    if event is not None:
        event = json.loads(event)
    else:
        event = {}

    if "G1" in event:
        new_row = { "G1": event["G1"], "G2": event["G2"],
                    "G3": event["G3"], "G4": event["G4"],
                    "G5": event["G5"], "G6": event["G6"],
                    "G7": event["G7"], "G8": event["G8"],
                    "G9": event["G9"], "G10": event["G10"]}

        new_x = pd.DataFrame.from_dict(new_row,
                                       orient = "index").transpose()
        prediction = str(model.predict_proba(new_x)[0][1])

        return { "body": "Prediction " + prediction }

    return { "body": "No parameters" }

```

Unlike Cloud Functions, Lambda functions authored in Python are not built on top of the Flask library. Instead of requiring a single parameter (`request`), a Lambda function requires `event` and `context` objects to be passed in as function parameters. The `event` includes the parameters of the request, and the `context` provides information about the execution environment of the function. When testing a Lambda function using the “Test” functionality in the Lambda console, the test configuration is passed directly to the function as a dictionary in the `event` object. However, when the function is called from the web, the `event` object is a dictionary that describes the web request, and the request parameters are stored in the `body` key in this dict. The first step in the Lambda function above checks if the function is being called directly from the console, or via the web. If the function is being called from the web, then the function overrides the `event` dictionary with the content in the `body` of the request.

One of the main differences from this approach with the GCP Cloud Function is that we did not need to explicitly define global variables that are lazily defined. With Lambda functions, you can define variables outside the scope of the function that are persisted before the function is invoked. It's important to load model objects outside of the model service function, because reloading the model each time a request is made can become expensive when handling large workloads.

To deploy the model, we need to create a zip file of the current directory, and upload the file to a location on S3. The snippet below shows how to perform these steps and then confirm that the upload succeeded using the `s3 ls` command. You'll need to modify the paths to use the S3 bucket name that you defined in the previous section.

```
zip -r logitFunction.zip .
aws s3 cp logitFunction.zip s3://dsp-ch3-logit/logitFunction.zip
aws s3 ls s3://dsp-ch3-logit/
```

Once your function is uploaded as a zip file to S3, you can return to the AWS console and set up a new Lambda function. Select “Author from scratch” as before, and under “Code entry type” select the option to upload from S3, specifying the location from the `cp` command above. You'll also need to define the `Handler`, which is a combination of the Python file name and the Lambda function name. An example configuration for the logit function is shown in Figure 3.6.

Make sure to select the Python runtime as the same version of Python that was used to run the `pip` commands on the EC2 instance. Once the function is deployed by pressing “Save”, we can test the function using the following definition for the test event.

```
{
  "G1": "1", "G2": "1", "G3": "1",
  "G4": "1", "G5": "1",
```

The screenshot shows the AWS Lambda console interface for a function named "logit". At the top, there are tabs for "Throttle", "Qualifiers", "Actions", and a dropdown menu currently set to "logit". To the right are "Test" and "Save" buttons. Below the tabs, the "Function code" section is active, showing a message: "The deployment package of your Lambda function 'logit' is too large to enable inline code editing. However, you can still invoke your function." Below this message, there are three dropdown menus: "Code entry type" set to "Upload a file from ...", "Runtime" set to "Python 3.6", and "Handler" set to "logit.lambda_handler". At the bottom, there is a section for "Amazon S3 link URL" with a text input field containing "s3://dsp-ch3-logit/logitFunction.zip".

FIGURE 3.6: Defining the logit function on AWS Lambda.

```
"G6": "1", "G7": "1", "G8": "1",
"G9": "1", "G10": "1"
}
```

Since the model is loaded when the function is deployed, the response time for testing the function should be relatively fast. An example output of testing the function is shown in Figure 3.7. The output of the function is a dictionary that includes a body key and the output of the model as the value. The function took 110 ms to execute and was billed for a duration of 200 ms.

So far, we’ve invoked the function only using the built-in test functionality of Lambda. In order to host the function so that other services can interact with the function, we’ll need to define an API Gateway. Under the “Designer” tab, click “Add Trigger” and select “API Gateway”. Next, select “Create a new API” and choose “Open” as the security setting. After setting up the trigger, an API Gateway should be visible in the Designer layout, as shown in Figure 3.8.

Before calling the function from Python code, we can use the API Gateway testing functionality to make sure that the function is set up properly. One of the challenges I ran into when testing this

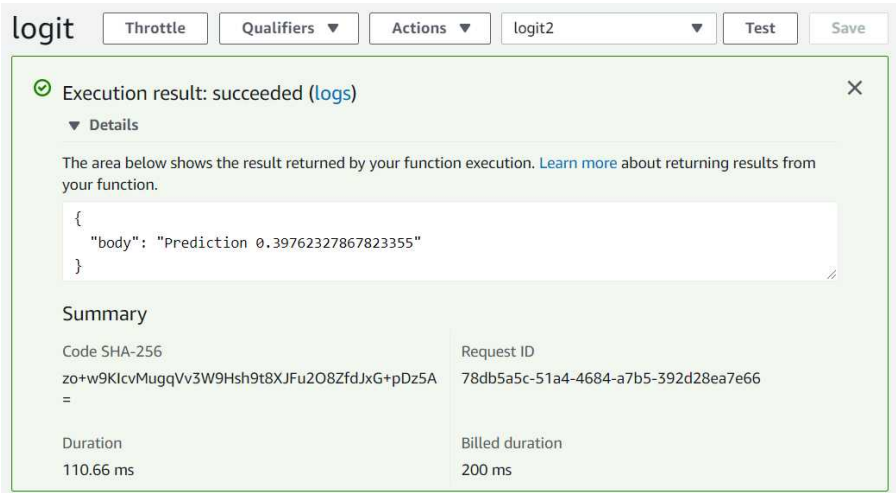


FIGURE 3.7: Testing the logit function on AWS Lambda.

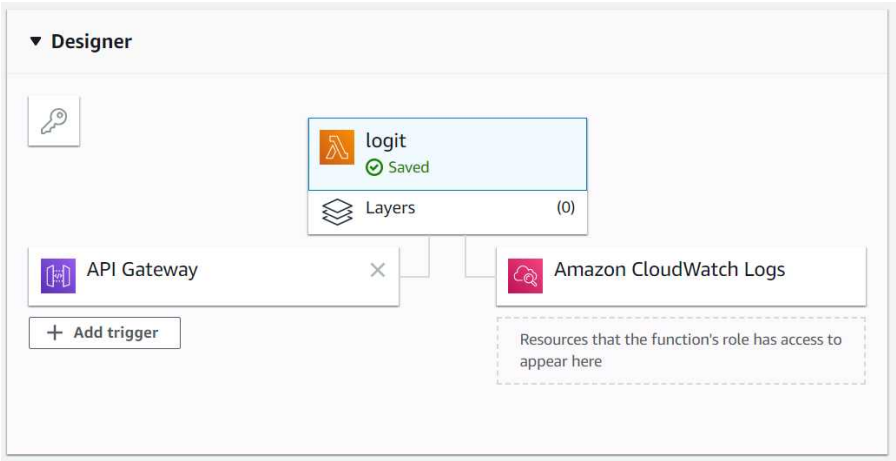


FIGURE 3.8: Setting up an API Gateway for the function.

Method Execution
/ logit - ANY - Method Test

Make a test call to your method with the provided input

Method

POST

Path

No path parameters exist for this resource. You can define path parameters by using the syntax {myPathParam} in a resource path.

Query Strings

{logit}

param1=value1¶m2=value2

Request: /logit

Status: 200

Latency: 101 ms

Response Body

Prediction 0.39762327867823355

Response Headers

{ "X-Amzn-Trace-Id": "Root=1-5d8f9544-7121cb984fd3f54959f53388;Sampled=0" }

Logs

FIGURE 3.9: Testing post commands on the Lambda function.

Lambda function was that the structure of the request varies when the function is invoked from the web versus the console. This is why the function first checks if the event object is a web request or dictionary with parameters. When you use the API Gateway to test the function, the resulting call will emulate calling the function as a web request. An example test of the logit function is shown in Figure 3.9.

Now that the gateway is set up, we can call the function from a remote host using Python. The code snippet below shows how to use a POST command to call the function and display the result. Since the function returns a string for the response, we use the text attribute rather than the json function to display the result.

```
import requests

result = requests.post("https://3z5btf0ucb.execute-api.us-east-1.
                        amazonaws.com/default/logit",
                        json = { 'G1':'1', 'G2':'0', 'G3':'0', 'G4':'0', 'G5':'0',
                                'G6':'0', 'G7':'0', 'G8':'0', 'G9':'0', 'G10':'0' })

print(result.text)
```

We now have a predictive model deployed to AWS Lambda that will autoscale as necessary to match workloads, and which requires minimal overhead to maintain.

Similar to Cloud Functions, there are a few different approaches that can be used to update the deployed models. However, for the approach we used in this section, updating the model requires updating the model file in the development environment, rebuilding the zip file and uploading it to S3, and then deploying a new version of the model. This is a manual process and if you expect frequent model updates, then it's better to rewrite the function so that it fetches the model definition from S3 directly rather than expecting the file to already be available in the local context. The most scalable approach is setting up additional triggers for the function, to notify the function that it's time to load a new model.

3.4 Conclusion

Serverless functions are a type of managed service that enable developers to deploy production-scale systems without needing to worry about infrastructure. To provide this abstraction, different cloud platforms do place constraints on how functions must be implemented, but the trade-off is generally worth the improvement in DevOps that these tools enable. While serverless technologies like Cloud Functions and Lambda can be operationally expensive, they provide flexibility that can offset these costs.

In this chapter, we implemented echo services and sklearn model endpoints using both GCP's Cloud Functions and AWS's Lambda offerings. With AWS, we created a local Python environment with all dependencies and then uploaded the resulting files to S3 to deploy functions, while in GCP we authored functions directly using the online code editor. The best system to use will likely depend on which cloud provider your organization is already using, but when prototyping new systems, it's useful to have hands on experience using more than one serverless function ecosystem.

Containers for Reproducible Models

When deploying data science models, it's important to be able to reproduce the same environment used for training and the environment used for serving. In Chapter 2, we used the same machine for both environments, and in Chapter 3 we used a `requirements.txt` file to ensure that the serverless ecosystem used for model serving matched our development environment. Container systems such as Docker provide a tool for building reproducible environments, and they are much lighter weight than alternative approaches such as virtual machines.

The idea of a container is that it is an isolated environment in which you can set up the dependencies that you need in order to perform a task. The task can be performing ETL work, serving ML models, standing up APIs, or hosting interactive web applications. The goal of a container framework is to provide isolation between instances with a lightweight footprint. With a container framework, you specify the dependencies that your code needs, and let the framework handle the legwork of managing different execution environments. Docker is the de facto standard for containers, and there is substantial tooling built this platform.

Elastic container environments, such as Elastic Container Service (ECS) provide similar functionality to serverless functions, where you want to abstract away the notion of servers from hosting data science models. The key differentiation is that serverless ecosystems are restricted to specific runtimes, often have memory limitations that make it challenging to use deep learning frameworks, and are cloud specific. With ECS, you are responsible for setting up the types of instances used to serve models, you can use whatever languages needed to serve the model, and you can take up

as much memory as needed. ECS still has the problem of being a proprietary AWS tool, but newer options such as EKS build upon Kubernetes which is open source and portable.

Here are some of the data science use cases I've seen for containers:

- **Reproducible Analyses:** Containers provide a great way of packaging up analyses, so that other team members can rerun your work months or years later.
- **Web Applications:** In Chapter 2 we built an interactive web application with Dash. Containers provide a great way of abstracting away hosting concerns for deploying the app.
- **Model Deployments:** If you want to expose your model as an endpoint, containers provide a great way of separating the model application code from model serving infrastructure.

The focus of this chapter will be the last use case. We'll take our web endpoint from Chapter 2 and wrap the application in a Docker container. We'll start by running the container locally on an EC2 instance, and then explore using ECS to create a scalable, load-balanced, and fault-tolerant deployment of our model. We'll then show how to achieve a similar result on GCP using Kubernetes.

Now that we are exploring scalable compute environments, it's important to keep an eye on cloud costs when using ECS and GKE. For AWS, it's useful to keep an eye on how many EC2 instances are provisioned, and on GCP the billing tool provides good tracking of costs. The section on orchestration is specific to AWS and uses an approach that is not portable to different cloud environments. Feel free to skip directly to the section on Kubernetes if AWS is not a suitable environment for your model deployments.

4.1 Docker

Docker, and other platform-as-a-service tools, provide a virtualization concept called containers. Containers run on top of a host operating system, but provide a standardized environment for code

running within the container. One of the key goals of this virtualization approach is that you can write code for a target environment, and any system running Docker can run your container.

Containers are a lightweight alternative to virtual machines, which provide similar functionality. The key difference is that containers are much faster to spin up, while providing the same level of isolation as virtual machines. Another benefit is that containers can re-use layers from other containers, making it much faster to build and share containers. Containers are a great solution to use when you need to run conflicting versions of Python runtimes or libraries, on a single machine.

With docker, you author a file called a *Dockerfile* that is used to define the dependencies for a container. The result of building the Dockerfile is a *Docker Image*, which packages all of the runtimes, libraries, and code needed to run an app. A *Docker Container* is an instantiated image that is running an application. One of the useful features in Docker is that new images can build off of existing images. For our model deployment, we'll extend the `ubuntu:latest` image.

This section will show how to set up Docker on an EC2 instance, author a Dockerfile for building an image of the echo service from Chapter 2, build an image using Docker, and run a container. To install Docker on an EC2 instance, you can use the `amazon-linux-extras` tool to simplify the process. The commands below will install Docker, start the service on the EC2 instance, and list the running Containers, which will return an empty list.

```
sudo yum install -y python3-pip python3 python3-setuptools
sudo yum update -y
sudo amazon-linux-extras install docker
sudo service docker start
sudo docker ps
```

The application we'll deploy is the echo service from Chapter 2. This service is a Flask application that parses the `msg` attribute

from a GET or POST and returns a JSON payload echoing the provided message. The only difference from the prior application is that the Flask app now runs on port 80, shown by the last line in the `echo.py` snippet below.

```
# load Flask
import flask
app = flask.Flask(__name__)

# define a predict function as an endpoint
@app.route("/predict", methods=["GET", "POST"])
def predict():
    data = {"success": False}

    # get the request parameters
    params = flask.request.json
    if (params == None):
        params = flask.request.args

    # if parameters are found, echo the msg parameter
    if (params != None):
        data["response"] = params.get("msg")
        data["success"] = True

    # return a response in json format
    return flask.jsonify(data)

# start the flask app, allow remote connections
app.run(host='0.0.0.0', port = 80)
```

Now that we have Docker installed and an application that we want to containerize, we need to write a Dockerfile that describes how to build an image. A Dockerfile for performing this task is shown in the snippet below. The first step is to use the `FROM` command to identify a base image to use. The `ubuntu` image provides a linux environment that supports the `apt-get` command. The `MAINTAINER` command adds to the metadata information associated with the

image, adding the name of the image maintainer. Next, the `RUN` command is used to install Python, set up a symbolic link, and install Flask. For containers with many Python libraries, it's also possible to use a `requirements.txt` file. The `COPY` command inserts our script into the image and places the file in the root directory. The final command specifies the arguments to run to execute the application.

```
FROM ubuntu:latest
MAINTAINER Ben Weber

RUN apt-get update \
    && apt-get install -y python3-pip python3-dev \
    && cd /usr/local/bin \
    && ln -s /usr/bin/python3 python \
    && pip3 install flask

COPY echo.py echo.py

ENTRYPOINT ["python3", "echo.py"]
```

After writing a `Dockerfile`, you can use the `build` command that `docker` provides to create an image. The first command shown in the snippet below shows how to build an image, tagged as `echo_service`, using the file `./Dockerfile`. The second command shows the list of Docker images available on the instance. The output will show both the `ubuntu` image we used as the base for our image, and our newly created image.

```
sudo docker image build -t "echo_service" .
sudo docker images
```

To run an image as a container, we can use the `run` command shown in the snippet below. The `-d` flag specifies that the container should run as a daemon process, which will continue to run even when shutting down the terminal. The `-p` flag is used to map a port on the host machine to a port that the container uses for

communication. Without this setting, our container is unable to receive external connections. The `ps` command shows the list of running containers, which should now include the echo service.

```
sudo docker run -d -p 80:80 echo_service
sudo docker ps
```

To test the container, we can use the same process as before where we use the external IP of the EC2 instance in a web browser and pass a `msg` parameter to the `/predict` endpoint. Since we set up a port mapping from the host port of 80 to the container port 80, we can directly invoke the container over the open web. An example invocation and result from the echo service container is shown below.

```
http://34.237.242.46/predict?msg=Hi_from_docker

{"response":"Hi_from_docker","success":true}
```

We have now walked through the process of building a Docker image and running the image as a container on an EC2 instance. While this approach does provide a solution for isolating different services on a machine, it does not provide scaling and fault-tolerance, which are typically requirements for a production-grade model deployment.

4.2 Orchestration

Container orchestration systems are responsible for managing the life cycles of containers in a cluster. They provide services including provisioning, scaling, failover, load balancing, and service discovery between containers. AWS has multiple orchestration solutions, but the general trend has been moving towards Kubernetes for this functionality, which is an open-source platform originally designed by Google.

One of the main reasons for using container orchestration as a data scientist is to be able to deploy models as containers, where you can scale up infrastructure to match demand, have a fault-tolerant system that can recover from errors, and have a static URL for your service managed by a load balancer. It's a bit of work to get one of these solutions up and running, but the end result is a robust model deployment. Serverless functions provide a similar capability, but using orchestration solutions means that you can use whatever programming language and runtime is necessary to serve your models. It provides flexibility at the cost of operational overhead, but as these tools evolve the overhead should be reduced.

The best solution to use for orchestration depends on your use case and if you have constraints on the cloud platform that you can use. If your goal is to use Kubernetes, then GCP provides a fully managed solution out of the box, which we'll cover in the next section. If you are restricted to AWS, then the two options are Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS). If you're just getting started with Kubernetes, then AWS provides less tools for getting containers up and running through a web console. Within ECS there are options for manually specifying EC2 instance types to run, or using the new Fargate mode to abstract away managing EC2 instances. My general recommendation is to learn Kubernetes if you want to get started with orchestration. But if you need to run a containerized model at scale in AWS, then ECS is currently the path of least friction.

In this section, we'll walk through deploying our model image to the AWS Container Registry, show how to set up a task in an ECS cluster to run the image as a container, and then set up a load-balanced service that provides managed provisioning for the echo service.

4.2.1 AWS Container Registry (ECR)

In order to use your Docker image in an orchestration system, you need to push your image to a Docker registry that works with the platform. For ECS, the AWS implementation of this service is called AWS Elastic Container Registry (ECR). ECR is a man-

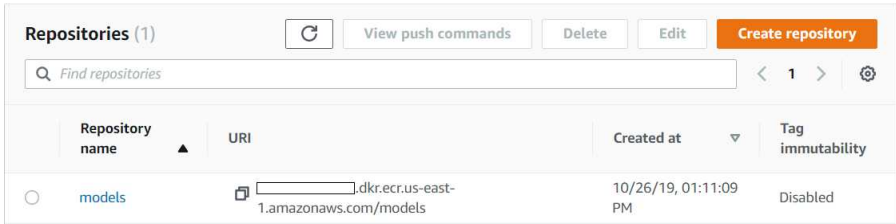


FIGURE 4.1: A model repository on ECR.

aged Docker registry that you can use to store and manage images within the AWS ecosystem. It works well with both ECS and EKS.

The goal of this subsection is to walk through the process of getting a Docker image from an EC2 instance to ECR. We'll cover the following steps:

1. Setting up an ECR repository
2. Creating an IAM role for ECR
3. Using docker login
4. Tagging an image
5. Pushing an image

The first step is to create a repository for the image that we want to store on ECR. A registry can have multiple repositories, and each repository can have multiple tagged images. To set up a new repository, perform the following steps from the AWS console:

1. Search for and select “ECR”
2. On the left panel, click on “Repositories”
3. Select “Create Repository”
4. Assign a repository name, e.g. “models”
5. Click on “Create Repository”

After completing these steps, you should have a new repository for saving images on ECR, as shown in Figure 4.1. The repository will initially be empty until we push a container.

Since our goal is to push a container from an EC2 instance to ECR, we'll need to set up permissions for pushing to the registry

from the command line. To set up these permissions, we can add additional policies to the `s3_lambda` user that we first created in Chapter 3. Perform the following steps from the AWS console:

1. Search for and select “IAM”
2. Click on Users
3. Select the “s3_lambda” user
4. Click “Add permissions”
5. Choose “AmazonEC2ContainerRegistryFullAccess”
6. Select “Attach existing policies directly”
7. Click on “Add permissions”

We now have permissions to write to ECR from the user account we previously set up. However, before we can run the `docker login` command for pushing images to ECR, we need to set up temporary credentials for accessing ECR. We can create a token by running the following command on the EC2 instance:

```
aws ecr get-login --region us-east-1 --no-include-email
```

The output of running this task is a command to run from the command line that will enable temporary access for storing images to ECR. For this to work on the EC2 instance, you’ll need to prepend `sudo` to the output from the following step and then run the generated command, as shown below.

```
sudo docker login -u AWS -p eyJwYXlsb2FkIjoiaV9vYWVrYnY0YlVqTFp...
```

If the authentication is successful, the output from this command should be `Login Succeeded`. Now that we have successfully used the `docker login` command, we can now push to the ECR repository. Before we can push our local image, we need to tag the image with account specific parameters. The snippet below shows how to tag the `echo` service with an AWS account ID and region, with a tag of `echo`. For these parameters, region is a string such as `us-east-1`, and the account ID is the number specified under `Account ID` in the “My Account” tab from the AWS console.

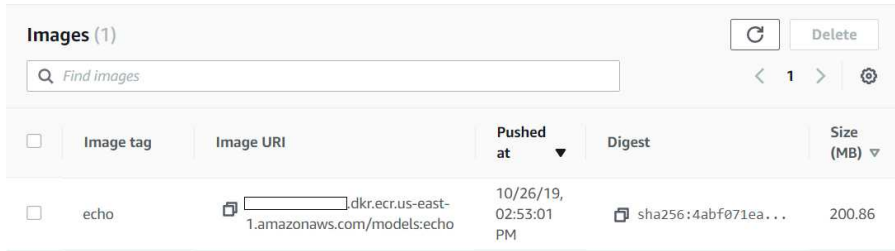


FIGURE 4.2: The echo image in the ECR repository.

```
sudo docker tag echo_service
[account_id].dkr.ecr.[region].amazonaws.com/models:echo
```

After tagging your image, it's good to check that the outcome matches the expected behavior. To check the tags of your images, run `sudo docker images` from the command line. An example output is shown below, with my account ID and region omitted.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
amazonaws.com/models	echo	3380f2a8805b	3 hours ago	473MB
echo_service	latest	3380f2a8805b	3 hours ago	473MB
ubuntu	latest	cf0f3ca922e0	8 days ago	64.2MB

The final step is to push the tagged image to the ECR repository. We can accomplish this by running the command shown below:

```
sudo docker push [account_id].dkr.[region].amazonaws.com/models:echo
```

After running this command, the echo service should now be available in the model repository on ECR. To check if the process succeeded, return to the AWS console and click on “Images” for the model repository. The repo should now show an image with the tag `models:echo`, as shown in Figure 4.2.

The outcome of this process is that we now have a Docker image pushed to ECR that can be leveraged by an orchestration system.

While we did walk through a number of AWS specific steps, the process of using Docker login applies to other cloud platforms.

4.2.2 AWS Container Service (ECS)

AWS provides an elastic container service called ECS that is a good platform for getting started with container management. While it is a proprietary technology, it does introduce many concepts that are useful to consider in an orchestrated cloud deployment. Here are some concepts exposed through an ECS deployment:

- **Cluster:** A cluster defines the environment used to provision and host containers.
- **Task:** A task is an instantiation of a container than performs a specific workload.
- **Service:** A service manages a task and provisions new machines based on demand.

Now that we have an image hosted on ECR, the next step is to use ECS to provide a scalable deployment of this image. In this section, we'll walk through the following steps:

1. Setting up a cluster
2. Setting up a task
3. Running a task
4. Running a service

At the end of this section, we'll have a service that manages a task running the echo service, but we'll connect directly to the IP of the provisioned EC2 instance. In the next section, we'll set up a load balancer to provide a static URL for accessing the service.

The first step in using ECS is to set up a service. There is a newer feature in ECS called Fargate that abstracts away the notion of EC2 instances when running a cluster, but this mode does not currently support the networking modes that we need for connecting directly to the container. To set up an ECS cluster, perform the following steps from the AWS console:

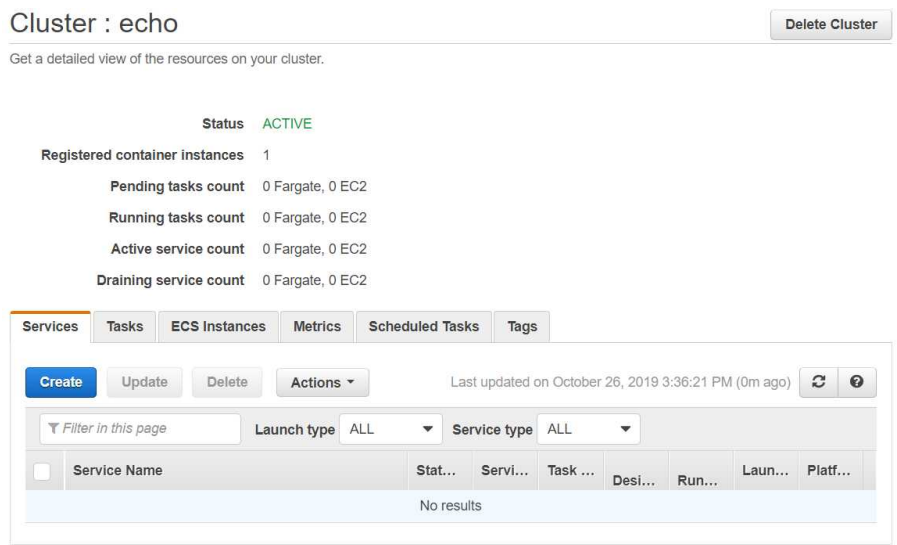


FIGURE 4.3: An empty ECS cluster.

- 1. Search for and select “ECS”
- 2. Click on “Clusters” on the left
- 3. Select “Create Cluster”
- 4. Use the “EC2 Linux + Networking” option
- 5. Assign a name, “dsp”
- 6. Select an instance type, m3.medium
- 7. For “VPC”, select an existing VPC
- 8. For “IAM Role”, use “aws-elasticbeanstalk-ec2-role”
- 9. Click “Create” to start the cluster

We now have an empty cluster with no tasks or services defined, as shown in Figure 4.3. While the cluster has not yet spun up any EC2 instances, there is an hourly billing amount associated with running the cluster, and any EC2 instances that do spin up will not be eligible for the free-tier option.

The next step is to define a task, which specifies the image to use and a number of different settings. A task can be executed directly, or managed via a service. Tasks are independent of services and a single task can be used across multiple services if necessary.

To set up a task, we'll first specify the execution environment by performing the following steps:

1. From the ECS console, click “Task Definitions”
2. Select “Create a new Task Definition”
3. Select EC2 launch type
4. Assign a name, “echo_task”
5. For “Task Role”, select “ecsTaskExecutionRole”
6. For “Network Mode”, use “Bridge”
7. Select “ecsTaskExecutionRole” for Task execution role
8. Set “Task Memory” and “Task CPU” to 1024

We then need to specify details about the Docker image that we want to host when running the task:

1. Click “Add container”
2. Assign a name “echo_service”
3. For “Image”, use the URI shown in Figure 4.2
4. Add a port mapping of host:80, container:80
5. Click “Add” to finalize container setup
6. Click “Create” to define the task

We now have a task setup in ECS that we can use to host our image as a container in the cloud. It is a good practice to test our your tasks in ECS before defining a service to manage the task. We can test out the task by performing the following steps:

1. From the ECS console, click on your cluster
2. Click on the “Tasks” tab
3. Select “Run new Task”
4. Use EC2 as the “Launch type”
5. For “Task Definition”, use “echo_task:1”
6. Click “Run Task”

The result of setting up and running the echo service as a task is shown in Figure 4.4. The ECS cluster is now running a single task, which is hosting our echo service. To provision this task, the ECS cluster will spin up a new EC2 instance. If you browse to

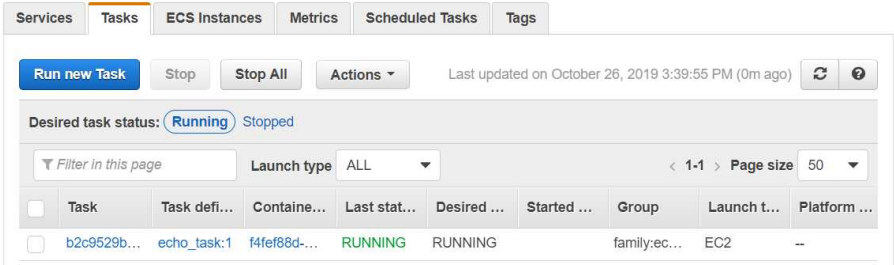


FIGURE 4.4: The echo task running on the ECS cluster.

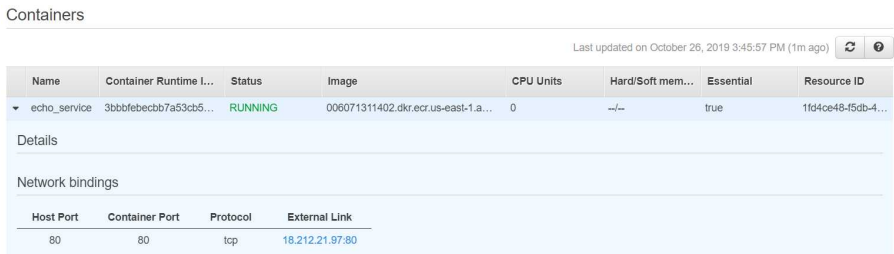


FIGURE 4.5: Network bindings for the echo service container.

the EC2 console in AWS, you'll see that a new EC2 instance has been provisioned, and the name of the instance will be based on the service, such as: ECS Instance - EC2ContainerService-echo.

Now that we have a container running in our ECS cluster, we can query it over the open web. To find the URL of the service, click on the running task and under containers, expand the echo service details. The console will show an external IP address where the container can be accessed, as shown in Figure 4.5. An example of using the echo service is shown in the snippet below.

```
http://18.212.21.97/predict?msg=Hi_from_ECS
```

```
{"response": "Hi_from_ECS", "success": true}
```

We now have a container running in the cloud, but it's not scalable and there is no fault tolerance. To enable these types of capabilities we need to define a service in our ECS cluster than manages the

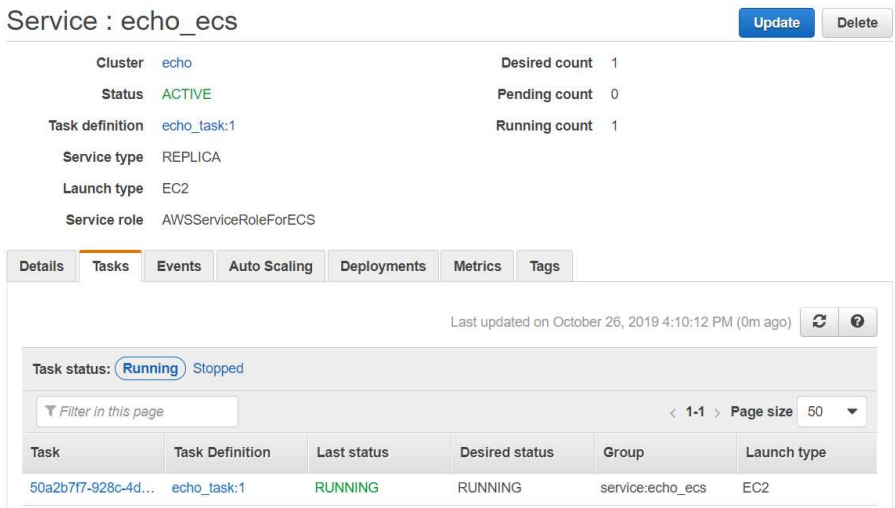


FIGURE 4.6: The ECS service running a single echo task.

lifecycle of a task. To set up a service for the task, perform the following steps:

1. From the ECS console, click on your cluster
2. Select the “Services” tab
3. Click “Create”
4. Use EC2 launch type
5. Select “echo_task” as the task
6. Assign a “Service name”, “echo_ecs”
7. Use “None” for load balancer type
8. Click “Create Service”

This will start the service. The service will set the “Desired count” value to 1, and it may take a few minutes for a new task to get ramped up by the cluster. Once “Running count” is set to 1, you can start using the service to host a model. An example of the provisioned service is shown in Figure 4.6. To find the IP of the container, click on the task within the service definition.

We now have a container that is managed by a service, but we’re still accessing the container directly. In order to use ECS in a way

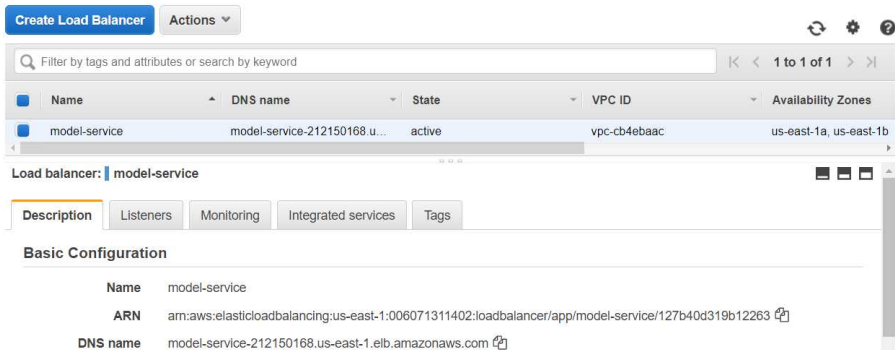


FIGURE 4.7: The Application Load Balancer configuration.

that scales, we need to set up a load balancer that provides a static URL and routes requests to active tasks managed by the service.

4.2.3 Load Balancing

There’s a number of different load balancer options in AWS that are useful for different deployments. This is also an area where the options are rapidly changing. For our ECS cluster, we can use the application load balancer to provide a static URL for accessing the echo service. To set up a load balancer, perform the following steps from the AWS console:

1. Search for and select “EC2”
2. Select “Load Balancer” on the left
3. Click “Create Load Balancer”
4. Choose “Application Load Balancer”
5. Assign a name, “model-service”
6. Use the default VPC
7. Create a new security group
8. Set source to “Anywhere”
9. Create a new target group, “model-group”
10. Click “Create”

An example of a provisioned application load balancer is shown in Figure 4.7. The last step needed to set up a load-balanced container is configuring an ECS service that uses the load balancer. Repeat

the same steps shown in the prior section for setting up a service, but instead of selecting “None” for the load balancer type, perform the following actions:

1. Select “Application” for Load Balance type
2. Select the “model-service” balancer
3. Select the “echo_service” container
4. Click “Add to Load Balancer”
5. Select “model-group” as the target group
6. Click “Create Service”

It’s taken quite a few steps, but our echo service is now running in a scalable environment, using a load balancer, and using a service that will manage tasks to handle failures and provision new EC2 instances as necessary. This approach is quite a bit of configuration versus Lambda for similar functionality, but this approach may be preferred based on the type of workload that you need to handle. There is cost involved with running an ECS cluster, even if you are not actively servicing requests, so understanding your expected workload is useful when modeling out the cost of different model serving options on AWS.

```
http://model123.us-east-1.elb.amazonaws.com/predict?msg=Hi_from_ELB
```

```
{"response": "Hi_from_ELB", "success": true}
```

AWS does provide an option for Kubernetes called EKS, but the options available through the web console are currently limited for managing Docker images. EKS can work with ECR as well, and as the AWS platform evolves EKS will likely be the best option for new deployments.

Make sure to terminate your cluster, load balancers, and EC2 instances once you are done testing out your deployment to reduce your cloud platform costs.

4.3 Kubernetes on GCP

Google Cloud Platform provides a service called Google Kubernetes Engine (GKE) for serving Docker containers. Kubernetes is a container-orchestration system originally developed by Google that is now open source. There are a wide range of use cases for this platform, but we'll focus on the specific task of hosting our echo service using managed Kubernetes.

Using Kubernetes for hosting a Docker container is similar to ECS, where the first step is to save your image to a Docker registry that can interface with the orchestration system. The GCP version of this registry service is called Container Registry. To get our image from an EC2 instance on AWS to the GCP Container Registry, we'll again use the docker login command. For this process to work, you'll need the GCP credentials json file that we set up in Chapter 1. The code snippet below shows how to pass the json file to the docker login command, tag the image for uploading it to the registry, and push the image to Container Registry.

```
cat dsdemo.json | sudo docker login -u _json_key
                                --password-stdin https://us.gcr.io
sudo docker tag echo_service us.gcr.io/[gcp_account]/echo_service
sudo docker push us.gcr.io/[gcp_account]/echo_service
```

You'll need to replace the `gcp_account` parameter in this script with your full google account ID. After performing these steps, the echo service image should be visible under the Registry view in the GCP console, as shown in Figure 4.8. Typically, if you are using GCP for serving models, it's likely that you'll be using Google Compute instances rather than EC2, but it's good to get practice interfacing between components in different cloud platforms.

The process for hosting a container with GKE is streamlined compared to all of the steps needed for using ECS. We'll first use the GCP console to set up a container on Kubernetes, and then expose

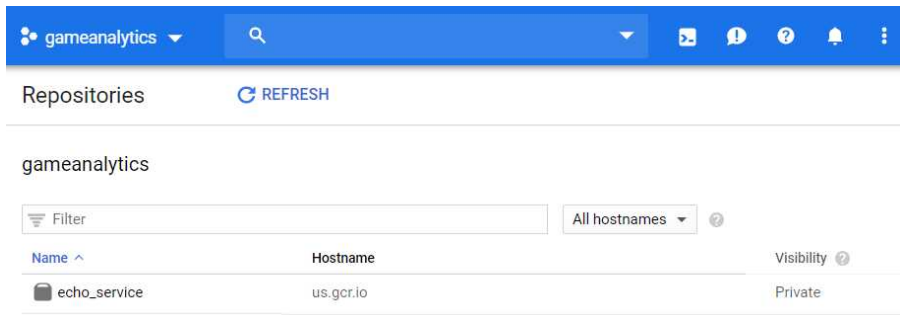


FIGURE 4.8: The echo image on GCP Container Registry.

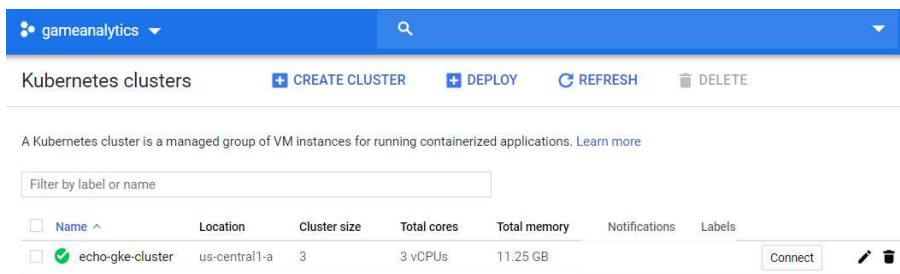


FIGURE 4.9: The echo image deployed via Kubernetes.

the service to the open web. To deploy the echo service container, perform the following steps from the GCP console:

1. Search for and select “Kubernetes Engine”
2. Click “Deploy Container”
3. Select “Existing Container Image”
4. Choose “echo_service:latest”
5. Assign an application name “echo-kge”
6. Click “Deploy”

We now have a Kubernetes cluster deployed and ready to serve the echo service. The deployment of a Kubernetes cluster on GKE can take a few minutes to set up. Once the deployment is completed, you should see the echo cluster under the list of clusters, as shown in Figure 4.9.

[←](#)
[Load balancer details](#)
[EDIT](#)
[DELETE](#)

a4ed35ac1f87611e987b842010a8000b

Frontend

Protocol ^	IP:Port	Network Tier ?
TCP	35.238.43.63:80	Premium

Backend

Name: a4ed35ac1f87611e987b842010a8000b Region: us-central1 Session affinity: None Health check: k8s-053e2bf8e96caf0c-node

Instances ^	35.238.43.63
gke-echo-gke-cluster-default-pool-41dc0b98-r71j	✓
gke-echo-gke-cluster-default-pool-41dc0b98-kp4v	✓
gke-echo-gke-cluster-default-pool-41dc0b98-jbv5	✓

FIGURE 4.10: The echo service deployed to the open web.

To use the service, we'll need to expose the cluster to the open web by performing the following steps from the GCP console:

1. From the GKE menu, select your cluster
2. Click on “Workloads”
3. Choose the “echo-gke” workload
4. Select the “Actions” tab and then “Expose”
5. For service type, select “load balancer”

After performing these steps, the cluster will configure an external IP that can be used to invoke the service, as shown in Figure 4.10. GKE will automatically load balance and scale the service as needed to match the workload.

```
http://35.238.43.63/predict?msg=Hi_from_GKE
```

```
{"response":"Hi_from_GKE","success":true}
```

An example of using the service is shown in the snippet above. We were able to quickly take a Docker image and deploy it in a Kubernetes ecosystem using GKE. It's good to build experience with Kubernetes for hosting Docker images, because it is a portable

solution that works across multiple cloud environments and it is being adopted by many open-source projects.

4.4 Conclusion

Containers are great to use to make sure that your analyses and models are reproducible across different environments. While containers are useful for keeping dependencies clean on a single machine, the main benefit is that they enable data scientists to write model endpoints without worrying about how the container will be hosted. This separation of concerns makes it easier to partner with engineering teams to deploy models to production, or using the approaches shown in this chapter, data and applied science teams can also own the deployment of models to production.

The best approach to use for serving models depends on your deployment environment and expected workload. Typically, you are constrained to a specific cloud platform when working at a company, because your model service may need to interface with other components in the cloud, such as a database or cloud storage. Within AWS, there are multiple options for hosting containers while GCP is aligned on GKE as a single solution. The main question to ask is whether it is more cost effective to serve your model using serverless function technologies or elastic container technologies. The correct answer will depend on the volume of traffic you need to handle, the amount of latency that is tolerable for end users, and the complexity of models that you need to host. Containerized solutions are great for serving complex models and making sure that you can meet latency requirements, but may require a bit more DevOps overhead versus serverless functions.

Workflow Tools for Model Pipelines

Model pipelines are usually part of a broader data platform that provides data sources, such as lakes and warehouses, and data stores, such as an application database. When building a pipeline, it's useful to be able to schedule a task to run, ensure that any dependencies for the pipeline have already completed, and to backfill historic data if needed. While it's possible to perform these types of tasks manually, there are a variety of tools that have been developed to improve the management of data science workflows.

In this chapter, we'll explore a batch model pipeline that performs a sequence of tasks in order to train and store results for a propensity model. This is a different type of task than the deployments we've explored so far, which have focused on serving real-time model predictions as a web endpoint. In a batch process, you perform a set of operations that store model results that are later served by a different application. For example, a batch model pipeline may predict which users in a game are likely to churn, and a game server fetches predictions for each user that starts a session and provides personalized offers.

When building batch model pipelines for production systems, it's important to make sure that issues with the pipeline are quickly resolved. For example, if the model pipeline is unable to fetch the most recent data for a set of users due to an upstream failure with a database, it's useful to have a system in place that can send alerts to the team that owns the pipeline and that can rerun portions of the model pipeline in order to resolve any issues with the prerequisite data or model outputs.

Workflow tools provide a solution for managing these types of problems in model pipelines. With a workflow tool, you specify

the operations that need to be completed, identify dependencies between the operations, and then schedule the operations to be performed by the tool. A workflow tool is responsible for running tasks, provisioning resources, and monitoring the status of tasks. There's a number of open source tools for building workflows including Airflow, Luigi, MLflow, and Pentaho Kettle. We'll focus on Airflow, because it is being widely adopted across companies and cloud platforms and are also providing fully-managed versions of Airflow.

In this chapter, we'll build a batch model pipeline that runs as a Docker container. Next, we'll schedule the task to run on an EC2 instance using cron, and then explore a managed version of cron using Kubernetes. In the third section, we'll use Airflow to define a graph of operations to perform in order to run our model pipeline, and explore a cloud offering of Airflow.

5.1 Sklearn Workflow

A common workflow for batch model pipelines is to extract data from a data lake or data warehouse, train a model on historic user behavior, predict future user behavior for more recent data, and then save the results to a data warehouse or application database. In the gaming industry, this is a workflow I've seen used for building likelihood to purchase and likelihood to churn models, where the game servers use these predictions to provide different treatments to users based on the model predictions. Usually libraries like sklearn are used to develop models, and languages such as PySpark are used to scale up to the full player base.

It is typical for model pipelines to require other ETLs to run in a data platform before the pipeline can run on the most recent data. For example, there may be an upstream step in the data platform that translates json strings into schematized events that are used as input for a model. In this situation, it might be necessary to rerun the pipeline on a day that issues occurred with the json

transformation process. For this section, we'll avoid this complication by using a static input data source, but the tools that we'll explore provide the functionality needed to handle these issues.

There's typically two types of batch model pipelines that can I've seen deployed in the gaming industry:

- **Persistent:** A separate training workflow is used to train models from the one used to build predictions. A model is persisted between training runs and loaded in the serving workflow.
- **Transient:** The same workflow is used for training and serving predictions, and instead of saving the model as a file, the model is rebuilt for each run.

In this section we'll build a transient batch pipeline, where a new model is retrained with each run. This approach generally results in more compute resources being used if the training process is heavyweight, but it helps avoid issues with model drift, which is useful to track. We'll author a pipeline that performs the following steps:

1. Fetches a data set from GitHub
2. Trains a logistic regression model
3. Applies the regression model
4. Saves the results to BigQuery

The pipeline will execute as a single Python script that performs all of these steps. For situations where you want to use intermediate outputs from steps across multiple tasks, it's useful to decompose the pipeline into multiple processes that are integrated through a workflow tool such as Airflow.

We'll build this workflow by first writing a Python script that runs on an EC2 instance, and then Dockerize the script so that we can use the container in workflows. To get started, we need to install a library for writing a Pandas dataframe to BigQuery:

```
pip install --user pandas_gbq
```

Next, we'll create a file called `pipeline.py` that performs the four pipeline steps identified above. The script shown below performs these steps by loading the necessary libraries, fetching the CSV file from GitHub into a Pandas dataframe, splits the dataframe into train and test groups to simulate historic and more recent users, builds a logistic regression model using the training data set, creates predictions for the test data set, and saves the resulting dataframe to BigQuery.

```
import pandas as pd
import numpy as np
from google.oauth2 import service_account
from sklearn.linear_model import LogisticRegression
from datetime import datetime
import pandas_gbq

# fetch the data set and add IDs
gamesDF = pd.read_csv("https://github.com/bgweber/Twitch/raw/
                      master/Recommendations/games-expand.csv")
gamesDF['User_ID'] = gamesDF.index
gamesDF['New_User'] = np.floor(np.random.randint(0, 10,
                                                  gamesDF.shape[0])/9)

# train and test groups
train = gamesDF[gamesDF['New_User'] == 0]
x_train = train.iloc[:,0:10]
y_train = train['label']
test = gamesDF[gamesDF['New_User'] == 1]
x_test = test.iloc[:,0:10]

# build a model
model = LogisticRegression()
model.fit(x_train, y_train)
y_pred = model.predict_proba(x_test)[: , 1]

# build a predictions dataframe
```

```

resultDF = pd.DataFrame({'User_ID':test['User_ID'], 'Pred':y_pred})
resultDF['time'] = str(datetime.now())

# save predictions to BigQuery
table_id = "dsp_demo.user_scores"
project_id = "gameanalytics-123"
credentials = service_account.Credentials.
                                from_service_account_file('dsdemo.json')
pandas_gbq.to_gbq(resultDF, table_id, project_id=project_id,
                  if_exists = 'replace', credentials=credentials)

```

To simulate a real-world data set, the script assigns a `User_ID` attribute to each record, which represents a unique ID to track different users in a system. The script also splits users into historic and recent groups by assigning a `New_User` attribute. After building predictions for each of the recent users, we create a results dataframe with the user ID, the model prediction, and a timestamp. It's useful to apply timestamps to predictions in order to determine if the pipeline has completed successfully. To test the model pipeline, run the following statements on the command line:

```

export GOOGLE_APPLICATION_CREDENTIALS=
    /home/ec2-user/dsdemo.json
python3 pipeline.py

```

If successfully, the script should create a new data set on BigQuery called `dsp_demo`, create a new table called `user_users`, and fill the table with user predictions. To test if data was actually populated in BigQuery, run the following commands in Jupyter:

```

from google.cloud import bigquery
client = bigquery.Client()

sql = "select * from dsp_demo.user_scores"
client.query(sql).to_dataframe().head()

```

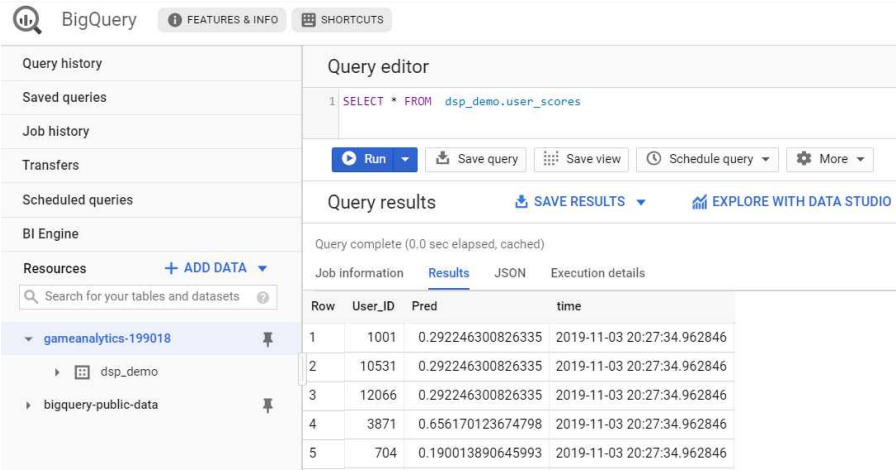


FIGURE 5.1: Querying the uploaded predictions in BigQuery.

This script will set up a client for connecting to BigQuery and then display the result set of the query submitted to BigQuery. You can also browse to the BigQuery web UI to inspect the results of the pipeline, as shown in Figure 5.1. We now have a script that can fetch data, apply a machine learning model, and save the results as a single process.

With many workflow tools, you can run Python code or bash scripts directly, but it’s good to set up isolated environments for executing scripts in order to avoid dependency conflicts for different libraries and runtimes. Luckily, we explored a tool for this in Chapter 4 and can use Docker with workflow tools. It’s useful to wrap Python scripts in Docker for workflow tools, because you can add libraries that may not be installed on the system responsible for scheduling, you can avoid issues with Python version conflicts, and containers are becoming a common way of defining tasks in workflow tools.

To containerize our workflow, we need to define a Dockerfile, as shown below. Since we are building out a new Python environment from scratch, we’ll need to install Pandas, sklearn, and the BigQuery library. We also need to copy credentials from the EC2 instance into the container so that we can run the `export` com-

mand for authenticating with GCP. This works for short term deployments, but for longer running containers it's better to run the export in the instantiated container rather than copying static credentials into images. The Dockerfile lists out the Python libraries needed to run the script, copies in the local files needed for execution, exports credentials, and specifies the script to run.

```
FROM ubuntu:latest
MAINTAINER Ben Weber

RUN apt-get update \
    && apt-get install -y python3-pip python3-dev \
    && cd /usr/local/bin \
    && ln -s /usr/bin/python3 python \
    && pip3 install pandas \
    && pip3 install sklearn \
    && pip3 install pandas_gbq

COPY pipeline.py pipeline.py
COPY /home/ec2-user/dsdemo.json dsdemo.json

RUN export GOOGLE_APPLICATION_CREDENTIALS=/dsdemo.json

ENTRYPOINT ["python3", "pipeline.py"]
```

Before deploying this script to production, we need to build an image from the script and test a sample run. The commands below show how to build an image from the Dockerfile, list the Docker images, and run an instance of the model pipeline image.

```
sudo docker image build -t "sklearn_pipeline" .
sudo docker images
sudo docker run sklearn_pipeline
```

After running the last command, the containerized pipeline should update the model predictions in BigQuery. We now have a model pipeline that we can run as a single bash command, which we

now need to schedule to run at a specific frequency. For testing purposes, we'll run the script every minute, but in practice models are typically executed hourly, daily, or weekly.

5.2 Cron

A common requirement for model pipelines is running a task at a regular frequency, such as every day or every hour. Cron is a utility that provides scheduling functionality for machines running the Linux operating system. You can Set up a scheduled task using the crontab utility and assign a cron expression that defines how frequently to run the command. Cron jobs run directly on the machine where cron is utilized, and can make use of the runtimes and libraries installed on the system.

There are a number of challenges with using cron in production-grade systems, but it's a great way to get started with scheduling a small number of tasks and it's good to learn the cron expression syntax that is used in many scheduling systems. The main issue with the cron utility is that it runs on a single machine, and does not natively integrate with tools such as version control. If your machine goes down, then you'll need to recreate your environment and update your cron table on a new machine.

A cron expression defines how frequently to run a command. It is a sequence of 5 numbers that define when to execute for different time granularities, and it can include wildcards to always run for certain time periods. A few sample expressions are shown in the snippet below:

```
# run every minute
* * * * *

# Run at 10am UTC everyday
0 10 * * *
```

```
# Run at 04:15 on Saturday
15 4 * * 6
```

When getting started with cron, it's good to use tools¹ to validate your expressions. Cron expressions are used in Airflow and many other scheduling systems.

We can use cron to schedule our model pipeline to run on a regular frequency. To schedule a command to run, run the following command on the console:

```
crontab -e
```

This command will open up the cron table file for editing in `vi`. To schedule the pipeline to run every minute, add the following commands to the file and save.

```
# run every minute
* * * * * sudo docker run sklearn_pipeline
```

After exiting the editor, the cron table will be updated with the new command to run. The second part of the cron statement is the command to run. When defining the command to run, it's useful to include full file paths. With Docker, we just need to define the image to run. To check that the script is actually executing, browse to the BigQuery UI and check the `time` column on the `user_scores` model output table.

We now have a utility for scheduling our model pipeline on a regular schedule. However, if the machine goes down then our pipeline will fail to execute. To handle this situation, it's good to explore cloud offerings with cron scheduling capabilities.

¹<https://crontab.guru/>

5.2.1 Cloud Cron

Both AWS and GCP provide managed cron options. With AWS it's possible to schedule services to run on ECS using cron expressions, and the GCP Kubernetes Engine provides scheduling support for containers. In this section, we'll explore the GKE option because it is simpler to set up. The first step is to push our model pipeline image to Container Registry using the following commands:

```
cat dsdemo.json | sudo docker login -u _json_key
                                --password-stdin https://us.gcr.io
sudo docker tag sklearn_pipeline
                                us.gcr.io/[gcp_account]/sklearn_pipeline
sudo docker push us.gcr.io/[gcp_account]/sklearn_pipeline
```

Next, we'll set up Kubernetes to schedule the image to run on a managed cluster. Unlike Section 4.3 where we created a cluster from an image directly, we'll use Kubernetes control (`kubectl`) commands to set up a scheduled container. Run the following commands from the GCP console to create a GKE cluster:

1. Search for and select “Kubernetes Engine”
2. Click “Create Cluster”
3. Select “Your first cluster”
4. Click “Create”

To schedule a task on this cluster, we'll use a YAML file to configure our task and then use a `kubectl` command to update the cluster with the model pipeline. Perform the following steps from the GKE UI to get access to the cluster and run a file editor:

1. Select the new cluster
2. Click on “Connect”
3. Select “Run in Cloud Shell”
4. Run the generated `gcloud` command
5. Run `vi sklearn.yaml`

This will provide terminal access to the console and allow us to save a YAML file on the cluster. The YAML file below shows how to define a task to run on a schedule with the specified cron expression.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: sklearn
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: sklearn
              image: us.gcr.io/[gcp_account]/sklearn_pipeline
              restartPolicy: OnFailure
```

After saving the file, we can use `kubectl` to update the cluster with the YAML file. Run the command below to update the cluster with the model pipeline task:

```
bgweber@cloudshell:~ (dsp)$ kubectl apply -f sklearn.yaml
cronjob.batch/sklearn created
```

To view the task, click on “workloads”. The events tab provides a log of the runs of the task, as shown in Figure 5.2. Again, we can validate that the pipeline is running successfully by browsing to the BigQuery UI and checking the `time` attribute of predictions.

There’s a variety of scheduling options for cloud platforms, and we explored just one option with GKE. GCP also provides a service called Cloud Scheduler, but this system does not work directly with Container Registry. Kubernetes is a good approach for handling scheduling in a cloud deployment, because the system is also

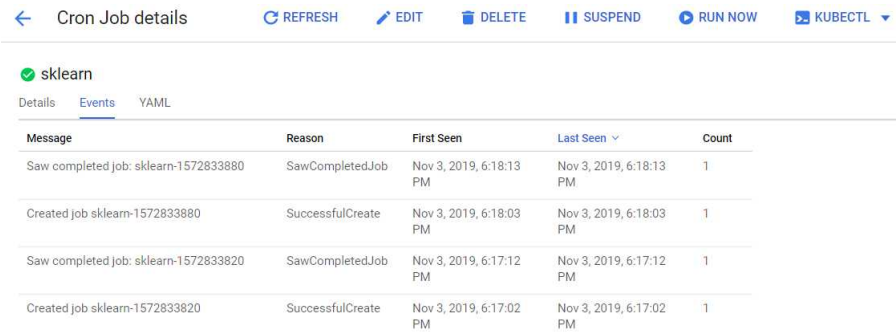


FIGURE 5.2: Execution events for the pipeline task on GKE.

responsible for provisioning machines that the scheduled task will use to execute.

5.3 Workflow Tools

Cron is useful for simple pipelines, but runs into challenges when tasks have dependencies on other tasks which can fail. To help resolve this issue, where tasks have dependencies and only portions of a pipeline need to be rerun, we can leverage workflow tools. Apache Airflow is currently the most popular tool, but other open source projects are available and provide similar functionality including Luigi and MLflow.

There are a few situations where workflow tools provide benefits over using cron directly:

- **Dependencies:** Workflow tools define graphs of operations, which makes dependencies explicit.
- **Backfills:** It may be necessary to run an ETL on old data, for a range of different dates.
- **Versioning:** Most workflow tools integrate with version control systems to manage graphs.
- **Alerting:** These tools can send out emails or generate PagerDuty alerts when failures occur.

Workflow tools are particularly useful in environments where different teams are scheduling tasks. For example, many game companies have data scientists that schedule model pipelines which are dependent on ETLs scheduled by a separate engineering team.

In this section, we'll schedule our task to run an EC2 instance using hosted Airflow, and then explore a fully-managed version of Airflow on GCP.

5.3.1 Apache Airflow

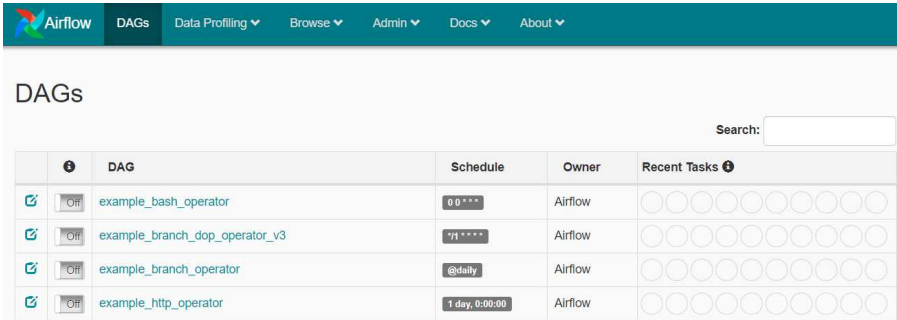
Airflow is an open source workflow tool that was originally developed by Airbnb and publically released in 2015. It helps solve a challenge that many companies face, which is scheduling tasks that have many dependencies. One of the core concepts in this tool is a graph that defines the tasks to perform and the relationships between these tasks.

In Airflow, a graph is referred to as a DAG, which is an acronym for directed acyclic graph. A DAG is a set of tasks to perform, where each task has zero or more upstream dependencies. One of the constraints is that cycles are not allowed, where two tasks have upstream dependencies on each other.

DAGs are set up using Python code, which is one of the differences from other workflow tools such as Pentaho Kettle which is GUI focused. The Airflow approach is called “configuration as code”, because a Python script defines the operations to perform within a workflow graph. Using code instead of a GUI to configure workflows is useful because it makes it much easier to integrate with version control tools such as GitHub.

To get started with Airflow, we need to install the library, initialize the service, and run the scheduler. To perform these steps, run the following commands on an EC2 instance or your local machine:

```
export AIRFLOW_HOME=~/.airflow
pip install --user apache-airflow
```



The screenshot shows the Airflow web interface. At the top is a navigation bar with the Airflow logo and links for DAGs, Data Profiling, Browse, Admin, Docs, and About. Below the navigation bar is a section titled 'DAGs' with a search bar. A table lists four example DAGs, each with a status icon, a name, a schedule, an owner, and a row of ten circles representing recent task status.

	DAG	Schedule	Owner	Recent Tasks
	example_bash_operator	0 0 * * *	Airflow	
	example_branch_dop_operator_v3	* * * * *	Airflow	
	example_branch_operator	@daily	Airflow	
	example_http_operator	1 day, 0:00:00	Airflow	

FIGURE 5.3: The Airflow web app running on an EC2 instance.

```
airflow initdb
airflow scheduler
```

Airflow also provides a web frontend for managing DAGs that have been scheduled. To start this service, run the following command in a new terminal on the same machine.

```
airflow webserver -p 8080
```

This command tells Airflow to start the web service on port 8080. You can open a web browser at this port on your machine to view the web frontend for Airflow, as shown in Figure 5.3.

Airflow comes preloaded with a number of example DAGs. For our model pipeline we'll create a new DAG and then notify Airflow of the update. We'll create a file called `sklearn.py` with the following DAG definition:

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime, timedelta

default_args = {
    'owner': 'Airflow',
    'depends_on_past': False,
```

```
'email': 'bgweber@gmail.com',
'start_date': datetime(2019, 11, 1),
'email_on_failure': True,
}

dag = DAG('games', default_args=default_args,
          schedule_interval="* * * * *")

t1 = BashOperator(
    task_id='sklearn_pipeline',
    bash_command='sudo docker run sklearn_pipeline',
    dag=dag)
```

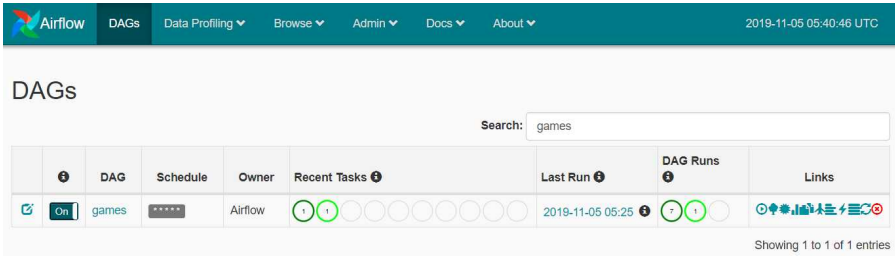
There's a few steps in this Python script to call out. The script uses a Bash operator to define the action to perform. The Bash operator is defined as the last step in the script, which specifies the operation to perform. The DAG is instantiated with a number of input arguments that define the workflow settings, such as who to email when the task fails. A cron expression is passed to the DAG object to define the schedule for the task, and the DAG object is passed to the Bash operator to associate the task with this graph of operations.

Before adding the DAG to airflow, it's useful to check for syntax errors in your code. We can run the following command from the terminal to check for issues with the DAG:

```
python3 sklearn.py
```

This command will not run the DAG, but will flag any syntax errors present in the script. To update Airflow with the new DAG file, run the following command:

```
airflow list_dags
```



	DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	games	*****	Airflow		2019-11-05 05:25		

Showing 1 to 1 of 1 entries

FIGURE 5.4: The sklearn DAG scheduled on Airflow.

DAGS

games

This command will add the DAG to the list of workflows in Airflow. To view the list of DAGs, navigate to the Airflow web server, as shown in Figure 5.4. The web server will show the schedule of the DAG, and provide a history of past runs of the workflow. To check that the DAG is actually working, browse to the BigQuery UI and check for fresh model outputs.

We now have an Airflow service up and running that we can use to monitor the execution of our workflows. This setup enables us to track the execution of workflows, backfill any gaps in data sets, and enable alerting for critical workflows.

Airflow supports a variety of operations, and many companies author custom operators for internal usage. In our first DAG, we used the Bash operator to define the task to execute, but other options are available for running Docker images, including the Docker operator. The code snippet below shows how to change our DAG to use the Docker operator instead of the Bash operator.

```
from airflow.operators.docker_operator import DockerOperator

t1 = DockerOperator(
    task_id='sklearn_pipeline',
```

```
image='sklearn_pipeline',  
dag=dag)
```

The DAG we defined does not have any dependencies, since the container performs all of the steps in the model pipeline. If we had a dependency, such as running a `sklearn_etl` container before running the model pipeline, we can use the `set_upstream` command as shown below. This configuration sets up two tasks, where the pipeline task will execute after the etl task completes.

```
t1 = BashOperator(  
    task_id='sklearn_etl',  
    bash_command='sudo docker run sklearn_etl',  
    dag=dag)  
  
t2 = BashOperator(  
    task_id='sklearn_pipeline',  
    bash_command='sudo docker run sklearn_pipeline',  
    dag=dag)  
  
t2.set_upstream(t1)
```

Airflow provides a rich set of functionality and we've only touched the surface of what the tool provides. While we were already able to schedule the model pipeline with hosted and managed cloud offerings, it's useful to schedule the task through Airflow for improved monitoring and versioning. The landscape of workflow tools will change over time, but many of the concepts of Airflow will translate to these new tools.

5.3.2 Managed Airflow

We now have a workflow tool set up for managing model workflows, but the default configuration does not provide high-availability, where new machines are provisioned when failures occur. While it's possible to set up a distributed version of Airflow using Celery to

set up different `Scheduler` and `Worker` nodes, one of the recent trends is using Kubernetes to create more robust Airflow deployments.

It is possible to self-host Airflow on Kubernetes, but it can be complex to set up. There are also fully-managed versions of Airflow available for cloud platforms such as Cloud Composer on GCP. With a managed version of Airflow, you define the DAGs to execute and set the schedules, and the platform is responsible for providing a high-availability deployment.

To run our DAG on Cloud Composer, we'll need to update the task to use a `GKEPodOperator` in place of a `DockerOperator`, because Composer needs to be able to authenticate with Container Registry. The updated DAG is shown in the snippet below.

```
from airflow.gcp.operators.kubernetes_engine import GKEPodOperator

t1 = GKEPodOperator(
    task_id='sklearn_pipeline',
    project_id = '{your_project_id}',
    cluster_name = ' us-central1-models-13d59d5b-gke',
    name = 'sklearn_pipeline',
    namespace='default',
    location='us-central1-c',
    image='us.gcr.io/{your_project_id}/sklearn_pipeline',
    dag=dag)
```

Cloud Composer is built on top of GKE. A beta version was released in 2018 and many features of the tool are still evolving. To get started with Composer, perform the following steps in the GCP Console:

1. Browse to Cloud Composer²
2. Click “Create”
3. Assign a name, “models”
4. Select Python 3

²<https://console.cloud.google.com/composer/>

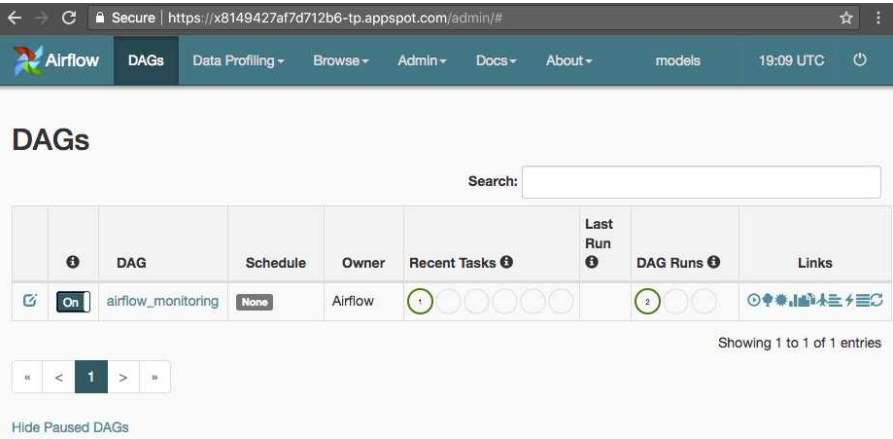


FIGURE 5.5: Fully-managed Airflow on Cloud Composer.

- 5. Select The most recent image version
- 6. Click “Create”

Like GKE, cluster setup takes quite a while to complete. Once setup completes, you can access the Airflow web service by clicking on “Airflow” in the list of clusters, as shown in Figure 5.5.

Before adding our DAG, we’ll need to update the `cluster_name` attribute to point to the provisioned GKE cluster. To find the cluster name, click on the Composer cluster and find the GKE cluster attribute. To add the DAG, click on “DAGs” in the cluster list, which will direct your browser to a google storage bucket. After uploading the DAG to this bucket using the upload UI, the Airflow cluster will automatically detect the new DAG and add it to the list of workflows managed by this cluster. We now have a high-availability Airflow deployment running our model workflow.

5.4 Conclusion

In this chapter we explored a batch model pipeline for applying a machine learning model to a set of users and storing the results to

BigQuery. To make the pipeline portable, so that we can execute it in different environments, we created a Docker image to define the required libraries and credentials for the pipeline. We then ran the pipeline on an EC2 instance using batch commands, cron, and Airflow. We also used GKE and Cloud Composer to run the container via Kubernetes.

Workflow tools can be tedious to set up, especially when installing a cluster deployment, but they provide a number of benefits over manual approaches. One of the key benefits is the ability to handle DAG configuration as code, which enables code reviews and version control for workflows. It's useful to get experience with configuration as code, because more and more projects are using this approach.

6

PySpark for Batch Pipelines

Spark is a general-purpose computing framework that can scale to massive data volumes. It builds upon prior big data tools such as Hadoop and MapReduce, while providing significant improvements in the expressivity of the languages it supports. One of the core components of Spark is resilient distributed datasets (RDD), which enable clusters of machines to perform workloads in a coordinated, and fault-tolerant manner. In more recent versions of Spark, the Dataframe API provides an abstraction on top of RDDs that resembles the same data structure in R and Pandas. PySpark is the Python interface to Spark, and it provides an API for working with large-scale datasets in a distributed computing environment.

PySpark is an extremely valuable tool for data scientists, because it can streamline the process for translating prototype models into production-grade model workflows. At Zynga, our data science team owns a number of production-grade systems that provide useful signals to our game and marketing teams. By using PySpark, we've been able to reduce the amount of support we need from engineering teams to scale up models from concept to production.

Up until now in this book, all of the models we've built and deployed have been targeted at single machines. While we are able to scale up model serving to multiple machines using Lambda, ECS, and GKS, these containers worked in isolation and there was no coordination among nodes in these environments. With PySpark, we can build model workflows that are designed to operate in cluster environments for both model training and model serving. The result is that data scientists can now tackle much larger-scale problems than previously possible using prior Python tools. PyS-

park provides a nice tradeoff between an expressive programming language and APIs to Spark versus more legacy options such as MapReduce. A general trend is that the use of Hadoop is dropping as more data science and engineering teams are switching to Spark ecosystems. In Chapter 7 we'll explore another distributed computing ecosystem for data science called Cloud Dataflow, but for now Spark is the open-source leader in this space. PySpark was one of the main motivations for me to switch from R to Python for data science workflows.

The goal of this chapter is to provide an introduction to PySpark for Python programmers that shows how to build large-scale model pipelines for batch scoring applications, where you may have billions of records and millions of users that need to be scored. While production-grade systems will typically push results to application databases, in this chapter we'll focus on batch processes that pull in data from a lake and push results back to the data lake for other systems to use. We'll explore pipelines that perform model applications for both AWS and GCP. While the data sets used in this chapter rely on AWS and GCP for storage, the Spark environment does not have to run on either of these platforms and instead can run on Azure, other clouds, or on-prem Spark clusters.

We'll cover a variety of different topics in this chapter to show different use cases of PySpark for scalable model pipelines. After showing how to make data available to Spark on S3, we'll cover some of the basics of PySpark focusing on dataframe operations. Next, we'll build out a predictive model pipeline that reads in data from S3, performs batch model predictions, and then writes the results to S3. We'll follow this by showing off how a newer feature called Pandas UDFs can be used with PySpark to perform distributed deep learning and feature engineering. To conclude, we'll build another batch model pipeline now using GCP and then discuss how to productize workflows in a Spark ecosystem.

6.1 Spark Environments

There's a variety of ways to both configure Spark clusters and submit commands to a cluster for execution. When getting started with PySpark as a data scientist, my recommendation is to use a freely-available notebook environment for getting up and running with Spark as quick as possible. While PySpark may not perform quite as well as Java or Scala for large-scale workflows, the ease of development in an interactive programming environment is worth the trade-off.

Based on your organization, you may be starting from scratch for Spark or using an existing solution. Here are the types of Spark deployments I've seen in practice:

- **Self Hosted:** An engineering team manages a set of clusters and provides console and notebook access.
- **Cloud Solutions:** AWS provides a managed Spark option called EMR and GCP has Cloud Dataproc.
- **Vendor Solutions:** Databricks, Cloudera, and other vendors provide fully-managed Spark environments.

There's a number of different factors to consider when choosing a Spark ecosystem, including cost, scalability, and feature sets. As you scale the size of the team using Spark, additional considerations are whether an ecosystem supports multi-tenancy, where multiple jobs can run concurrently on the same cluster, and isolation where one job failing should not kill other jobs. Self-hosted solutions require significant engineering work to support these additional considerations, so many organizations use cloud or vendor solutions for Spark. In this book, we'll use the Databricks Community Edition, which provides all of the baseline features needed for learning Spark in a collaborative notebook environment.

Spark is a rapidly evolving ecosystem, and it's difficult to author books about this subject that do not quickly become out of date as the platform evolves. Another issue is that many books target Scala rather than Python for the majority of coding examples. My

advice for readers that want to dig deeper into the Spark ecosystem is to explore books based on the broader Spark ecosystem, such as (Karau et al., 2015). You'll likely need to read through Scala or Java code examples, but the majority of content covered will be relevant to PySpark.

6.1.1 Spark Clusters

A Spark environment is a cluster of machines with a single driver node and zero or more worker nodes. The driver machine is the master node in the cluster and is responsible for coordinating the workloads to perform. In general, workloads will be distributed across the worker nodes when performing operations on Spark dataframes. However, when working with Python objects, such as lists or dictionaries, objects will be instantiated on the driver node.

Ideally, you want all of your workloads to be operating on worker nodes, so that the execution of the steps to perform is distributed across the cluster, and not bottlenecked by the driver node. However, there are some types of operations in PySpark where the driver has to perform all of the work. The most common situation where this happens is when using Pandas dataframes in your workloads. When you use `toPandas` or other commands to convert a data set to a Pandas object, all of the data is loaded into memory on the driver node, which can crash the driver node when working with large data sets.

In PySpark, the majority of commands are lazily executed, meaning that an operation is not performed until an output is explicitly needed. For example, a `join` operation between two Spark dataframes will not immediately cause the join operation to be performed, which is how Pandas works. Instead, the join is performed once an output is added to the chain of operations to perform, such as displaying a sample of the resulting dataframe. One of the key differences between Pandas operations, where operations are eagerly performed and pulled into memory, is that PySpark operations are lazily performed and not pulled into memory until needed. One of the benefits of this approach is that the graph of

operations to perform can be optimized before being sent to the cluster to execute.

In general, nodes in a Spark cluster should be considered ephemeral, because a cluster can be resized during execution. Additionally, some vendors may spin up a new cluster when scheduling a job to run. This means that common operations in Python, such as saving files to disk, do not map directly to PySpark. Instead, using a distributed computing environment means that you need to use a persistent file store such as S3 when saving data. This is important for logging, because a worker node may crash and it may not be possible to `ssh` into the node for debugging. Most Spark deployments have a logging system set up to help with this issue, but it's good practice to log workflow status to persistent storage.

6.1.2 Databricks Community Edition

One of the quickest ways to get up and running with PySpark is to use a hosted notebook environment. Databricks is the largest Spark vendor and provides a free version for getting started called Community Edition¹. We'll use this environment to get started with Spark and build AWS and GCP model pipelines.

The first step is to create a login on the Databricks website for the community edition. Next, perform the following steps to spin up a test cluster after logging in:

1. Click on “Clusters” on the left navigation bar
2. Click “Create Cluster”
3. Assign a name, “DSP”
4. Select the most recent runtime (non-beta)
5. Click “Create Cluster”

After a few minutes we'll have a cluster set up that we can use for submitting Spark commands. Before attaching a notebook to the cluster, we'll first set up the libraries that we'll use throughout

¹<https://community.cloud.databricks.com/>

this chapter. Instead of using `pip` to install libraries, we'll use the Databricks UI, which makes sure that every node in the cluster has the same set of libraries installed. We'll use both Maven and PyPI to install libraries on the cluster. To install the BigQuery connector, perform the following steps:

1. Click on “Clusters” on the left navigation bar
2. Select the “DSP” cluster
3. Click on the “Libraries” tab
4. Select “Install New”
5. Click on the “Maven” tab.
6. Set coordinates to `com.spotify:spark-bigquery_2.11:0.2.2`
7. Click install

The UI will then show the status as resolving, and then installing, and then installed. We also need to attach a few Python libraries that are not pre-installed on a new Databricks cluster. Standard libraries such as Pandas are installed, but you might need to upgrade to a more recent version since the libraries pre-installed by Databricks can lag significantly.

To install a Python library on Databricks, perform the same steps as before up to step 5. Next, instead of selecting “Maven” choose “PyPI”. Under `Package`, specify the package you want to install and then click “Install”. To follow along with all of the sections in this chapter, you'll need to install the following Python packages:

- **koalas** - for dataframe conversion.
- **featuretools** - for feature generation.
- **tensorflow** - for a deep learning backend.
- **keras** - for a deep learning model.

You'll now have a cluster set up capable of performing distributed feature engineering and deep learning. We'll start with basic Spark commands, show off newer functionality such as the `koalas` library, and then dig into these more advanced topics. After this setup, your cluster library setup should look like Figure 6.1. To ensure that everything is set up successfully, restart the cluster and check the status of the installed libraries.

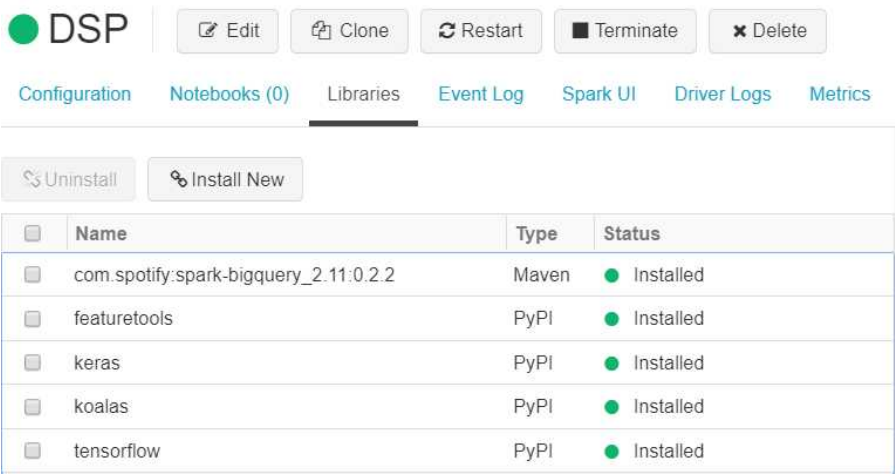


FIGURE 6.1: Libraries attached to a Databricks cluster.

Now that we have provisioned a cluster and set up the required libraries, we can create a notebook to start submitting commands to the cluster. To create a new notebook, perform the following steps:

1. Click on “Databricks” on the left navigation bar
2. Under “Common Tasks”, select “New Notebook”
3. Assign a name “CH6”
4. Select “Python” as the language
5. select “DSP” as the cluster
6. Click “Create”

The result will be a notebook environment where you can start running Python and PySpark commands, such as `print("Hello World!")`. An example notebook running this command is shown in Figure 6.2. We now have a PySpark environment up and running that we can use to build distributed model pipelines.

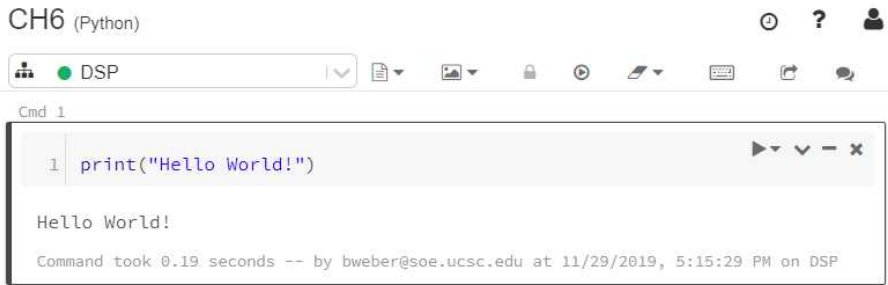


FIGURE 6.2: Running a Python command in Databricks.

6.2 Staging Data

Data is essential for PySpark workflows. Spark supports a variety of methods for reading in data sets, including connecting to data lakes and data warehouses, as well as loading sample data sets from libraries, such as the Boston housing data set. Since the theme of this book is building scalable pipelines, we'll focus on using data layers that work with distributed workflows. To get started with PySpark, we'll stage input data for a model pipeline on S3, and then read in the data set as a Spark dataframe.

This section will show how to stage data to S3, set up credentials for accessing the data from Spark, and fetching the data from S3 into a Spark dataframe. The first step is to set up a bucket on S3 for storing the data set we want to load. To perform this step, run the following operations on the command line.

```
aws s3api create-bucket --bucket dsp-ch6 --region us-east-1
aws s3 ls
```

After running the command to create a new bucket, we use the `ls` command to verify that the bucket has been successfully created. Next, we'll download the games data set to the EC2 instance and then move the file to S3 using the `cp` command, as shown in the snippet below.

```
wget https://github.com/bgweber/Twitch/raw/master/  
Recommendations/games-expand.csv  
aws s3 cp games-expand.csv s3://dsp-ch6/csv/games-expand.csv
```

In addition to staging the games data set to S3, we'll also copy a subset of the CSV files from the Kaggle NHL data set, which we set up in Section 1.5.2. Run the following commands to stage the plays and stats CSV files from the NHL data set to S3.

```
aws s3 cp game_plays.csv s3://dsp-ch6/csv/game_plays.csv  
aws s3 cp game_skater_stats.csv  
s3://dsp-ch6/csv/game_skater_stats.csv  
aws s3 ls s3://dsp-ch6/csv/
```

We now have all of the data sets needed for the code examples in this chapter. In order to read in these data sets from Spark, we'll need to set up S3 credentials for interacting with S3 from the Spark cluster.

6.2.1 S3 Credentials

For production environments, it is better to use IAM roles to manage access instead of using access keys. However, the community edition of Databricks constrains how much configuration is allowed, so we'll use access keys to get up and running with the examples in this chapter. We already set up a user for accessing S3 from an EC2 instance. To create a set of credentials for accessing S3 programmatically, perform the following steps from the AWS console:

1. Search for and select "IAM"
2. Click on "Users"
3. Select the user created in Section 3.3.2, "S3_Lambda"
4. Click "Security Credentials"
5. Click "Create Access Key"

The result will be an access key and a secret key enabling access to S3. Save these values in a secure location, as we'll use them in

the notebook to connect to the data sets on S3. Once you are done with this chapter, it is recommended to revoke these credentials.

Now that we have credentials set up for access, we can return to the Databricks notebook to read in the data set. To enable access to S3, we need to set the access key and secret key in the Hadoop configuration of the cluster. To set these keys, run the PySpark commands shown in the snippet below. You'll need to replace the access and secret keys with the credentials we just created for the `S3_Lambda` role.

```
AWS_ACCESS_KEY = "AK..."
AWS_SECRET_KEY = "dC..."

sc._jsc.hadoopConfiguration().set(
    "fs.s3n.awsAccessKeyId", AWS_ACCESS_KEY)
sc._jsc.hadoopConfiguration().set(
    "fs.s3n.awsSecretAccessKey", AWS_SECRET_KEY)
```

We can now read the data set into a Spark dataframe using the `read` command, as shown below. This command uses the `spark` context to issue a read command and reads in the data set using the CSV input reader. We also specify that the CSV file includes a header row and that we want Spark to infer the data types for the columns. When reading in CSV files, Spark eagerly fetches the data set into memory, which can cause issues for larger data sets. When working with large CSV files, it's a best practice to split up large data sets into multiple files and then read in the files using a wildcard in the input path. When using other file formats, such as Parquet or Avro, Spark lazily fetches the data sets.

```
games_df = spark.read.csv("s3://dsp-ch6/csv/games-expand.csv",
                           header=True, inferSchema = True)

display(games_df)
```

The `display` command in the snippet above is a utility function provided by Databricks that samples the input dataframe and shows

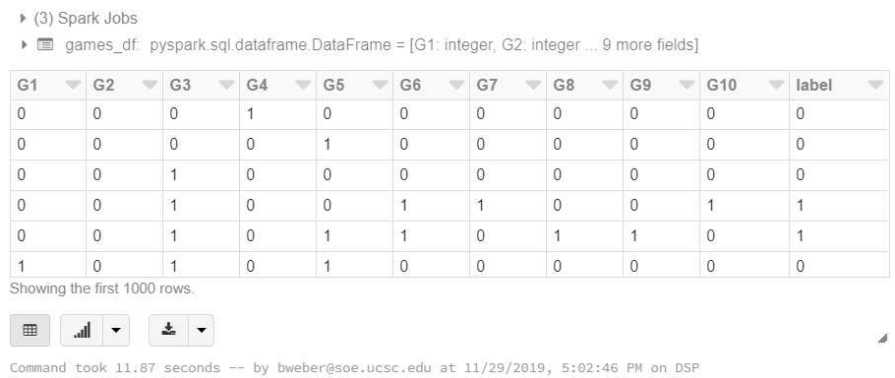


FIGURE 6.3: Displaying the dataframe in Databricks.

a table representation of the frame, as shown in Figure 6.3. It is similar to the `head` function in Pandas, but provides additional functionality such as transforming the sampled dataframe into a plot. We'll explore the plotting functionality in Section 6.3.3.

Now that we have data loaded into a Spark dataframe, we can begin exploring the PySpark language, which enables data scientists to build production-grade model pipelines.

6.3 A PySpark Primer

PySpark is a powerful language for both exploratory analysis and building machine learning pipelines. The core data type in PySpark is the Spark dataframe, which is similar to Pandas dataframes, but is designed to execute in a distributed environment. While the Spark Dataframe API does provide a familiar interface for Python programmers, there are significant differences in the way that commands issued to these objects are executed. A key difference is that Spark commands are lazily executed, which means that commands such as `iloc` are not available on these objects. While working with Spark dataframes can seem constraining, the benefit is that PySpark can scale to much larger data sets than Pandas.

This section will walk through common operations for Spark dataframes, including persisting data, converting between different dataframe types, transforming dataframes, and using user-defined functions. We'll use the NHL stats data set, which provides user-level summaries of player performance for each game. To load this data set as a Spark dataframe, run the commands in the snippet below.

```
stats_df = spark.read.csv("s3://dsp-ch6/csv/game_skater_stats.csv",
                           header=True, inferSchema = True)
display(stats_df)
```

6.3.1 Persisting Dataframes

A common operation in PySpark is saving a dataframe to persistent storage, or reading in a data set from a storage layer. While PySpark can work with databases such as Redshift, it performs much better when using distributed file stores such as S3 or GCS. In this chapter we'll use these types of storage layers as the outputs of model pipelines, but it's also useful to stage data to S3 as intermediate steps within a workflow. For example, in the AutoModel² system at Zynga, we stage the output of the feature generation step to S3 before using MLlib to train and apply machine learning models.

The data storage layer to use will depend on your cloud platform. For AWS, S3 works well with Spark for distributed data reads and writes. When using S3 or other data lakes, Spark supports a variety of different file formats for persisting data. Parquet is typically the industry standard when working with Spark, but we'll also explore Avro and ORC in addition to CSV. Avro is a better format for streaming data pipelines, and ORC is useful when working with legacy data pipelines.

To show the range of data formats supported by Spark, we'll take the stats data set and write it to Avro, then Parquet, then ORC,

²<https://www.gamasutra.com/blogs/BenWeber/20190426/340293/>

and finally CSV. After performing this round trip of data IO, we'll end up with our initial Spark dataframe. To start, we'll save the stats dataframe in Avro format, using the code snippet shown below. This code writes the dataframe to S3 in Avro format using the Databricks Avro writer, and then reads in the results using the same library. The result of performing these steps is that we now have a Spark dataframe pointing to the Avro files on S3. Since PySpark lazily evaluates operations, the Avro files are not pulled to the Spark cluster until an output needs to be created from this data set.

```
# AVRO write
avro_path = "s3://dsp-ch6/avro/game_skater_stats/"
stats_df.write.mode('overwrite').format(
    "com.databricks.spark.avro").save(avro_path)

# AVRO read
avro_df = sqlContext.read.format(
    "com.databricks.spark.avro").load(avro_path)
```

Avro is a distributed file format that is record based, while the Parquet and ORC formats are column based. It is useful for the streaming workflows that we'll explore in Chapter 8, because it compresses records for distributed data processing. The output of saving the stats dataframe in Avro format is shown in the snippet below, which shows a subset of the status files and data files generated when persisting a dataframe to S3 as Avro. Like most scalable data formats, Avro will write records to several files, based on partitions if specified, in order to enable efficient read and write operations.

```
aws s3 ls s3://dsp-ch6/avro/game_skater_stats/
2019-11-27 23:02:43      1455 _committed_1588617578250853157
2019-11-27 22:36:31      1455 _committed_1600779730937880795
2019-11-27 23:02:40         0 _started_1588617578250853157
2019-11-27 23:31:42         0 _started_6942074136190838586
```

```
2019-11-27 23:31:47      1486327 part-00000-tid-6942074136190838586-  
                        c6806d0e-9e3d-40fc-b212-61c3d45c1bc3-15-1-c000.avro  
2019-11-27 23:31:43      44514 part-00007-tid-6942074136190838586-  
                        c6806d0e-9e3d-40fc-b212-61c3d45c1bc3-22-1-c000.avro
```

Parquet on S3 is currently the standard approach for building data lakes on AWS, and tools such as Delta Lake are leveraging this format to provide highly-scalable data platforms. Parquet is a columnar-oriented file format that is designed for efficient reads when only a subset of columns are being accessed for an operation, such as when using Spark SQL. Parquet is a native format for Spark, which means that PySpark has built-in functions for both reading and writing files in this format.

An example of writing the stats dataframe as Parquet files and reading in the result as a new dataframe is shown in the snippet below. In this example, we haven't set a partition key, but as with Avro, the dataframe will be split up into multiple files in order to support highly-performant read and write operations. When working with large-scale data sets, it's useful to set partition keys for the file export using the `repartition` function. After this section, we'll use Parquet as the primary file format when working with Spark.

```
# parquet out  
parquet_path = "s3a://dsp-ch6/games-parquet/"  
avro_df.write.mode('overwrite').parquet(parquet_path)  
  
# parquet in  
parquet_df = sqlContext.read.parquet(parquet_path)
```

ORC is another columnar format that works well with Spark. The main benefit over Parquet is that it can support improved compression, at the cost of additional compute cost. I'm including it in this chapter, because some legacy systems still use this format. An example of writing the stats dataframe to ORC and reading the results back into a Spark dataframe is shown in the snippet below.

Like the Avro format, the ORC write command will distribute the dataframe to multiple files based on the size.

```
# orc out
orc_path = "s3a://dsp-ch6/games-orc/"
parquet_df.write.mode('overwrite').orc(orc_path)

# orc in
orc_df = sqlContext.read.orc(orc_path)
```

To complete our round trip of file formats, we'll write the results back to S3 in the CSV format. To make sure that we write a single file rather than a batch of files, we'll use the `coalesce` command to collect the data to a single node before exporting it. This is a command that will fail with large data sets, and in general it's best to avoid using the CSV format when using Spark. However, CSV files are still a common format for sharing data, so it's useful to understand how to export to this format.

```
# CSV out
csv_path = "s3a://dsp-ch6/games-csv-out/"
orc_df.coalesce(1).write.mode('overwrite').format(
    "com.databricks.spark.csv").option("header","true").save(csv_path)

# and CSV read to finish the round trip
csv_df = spark.read.csv(csv_path, header=True, inferSchema = True)
```

The resulting dataframe is the same as the dataframe that we first read in from S3, but if the data types are not trivial to infer, then the CSV format can cause problems. When persisting data with PySpark, it's best to use file formats that describe the schema of the data being persisted.

6.3.2 Converting Dataframes

While it's best to work with Spark dataframes when authoring PySpark workloads, it's often necessary to translate between dif-

ferent formats based on your use case. For example, you might need to perform a Pandas operation, such as selecting a specific element from a dataframe. When this is required, you can use the `toPandas` function to pull a Spark dataframe into memory on the driver node. The PySpark snippet below shows how to perform this task, display the results, and then convert the Pandas dataframe back to a Spark dataframe. In general, it's best to avoid Pandas when authoring PySpark workflows, because it prevents distribution and scale, but it's often the best way of expressing a command to execute.

```
stats_pd = stats_df.toPandas()

stats_df = sqlContext.createDataFrame(stats_pd)
```

To bridge the gap between Pandas and Spark dataframes, Databricks introduced a new library called Koalas that resembles the Pandas API for Spark-backed dataframes. The result is that you can author Python code that works with Pandas commands that can scale to Spark-scale data sets. An example of converting a Spark dataframe to Koalas and back to Spark is shown in the following snippet. After converting the stats dataframe to Koalas, the snippet shows how to calculate the average time on ice as well as index into the Koalas frame. The intent of Koalas is to provide a Pandas interface to Spark dataframes, and as the Koalas library matures more Python modules may be able to take advantage of Spark. The output from the snippet shows that the average time on ice was 993 seconds per game for NHL players.

```
import databricks.koalas as ks

stats_ks = stats_df.to_koalas()
stats_df = stats_ks.to_spark()

print(stats_ks['timeOnIce'].mean())
print(stats_ks.iloc[:1, 1:2])
```

During the development of this book, Koalas is still preliminary and only partially implemented, but it strives to provide a familiar interface for Python coders. Both Pandas and Spark dataframes can work with Koalas, and the snippet below shows how to go from Spark to Koalas to Pandas to Spark, and Spark to Pandas to Koalas to Spark.

```
# spark -> koalas -> pandas -> spark
df = sqlContext.createDataFrame(stats_df.to_koalas().toPandas())

# spark -> pandas -> koalas -> spark
df = ks.from_pandas(stats_df.toPandas()).to_spark()
```

In general, you'll be working with Spark dataframes when authoring code in a PySpark environment. However, it's useful to be able to work with different object types as necessary to build model workflows. Koalas and Pandas UDFs provide powerful tools for porting workloads to large-scale data ecosystems.

6.3.3 Transforming Data

The PySpark Dataframe API provides a variety of useful functions for aggregating, filtering, pivoting, and summarizing data. While some of this functionality maps well to Pandas operations, my recommendation for quickly getting up and running with munging data in PySpark is to use the SQL interface to dataframes in Spark called Spark SQL. If you're already using the `pandasql` or `framequery` libraries, then Spark SQL should provide a familiar interface. If you're new to these libraries, then the SQL interface still provides an approachable way of working with the Spark ecosystem. We'll cover the Dataframe API later in this section, but first start with the SQL interface to get up and running.

Exploratory data analysis (EDA) is one of the key steps in a data science workflow for understanding the shape of a data set. To work through this process in PySpark, we'll load the stats data set into a dataframe, expose it as a view, and then calculate summary statistics. The snippet below shows how to load the NHL stats data

player_id	games	goals
8471214	788	434
8474564	655	342
8474141	748	311
8475166	700	308
8470794	782	305

FIGURE 6.4: Summarizing player activity.

set, expose it as a view to Spark, and then run a query against the dataframe. The aggregated dataframe is then visualized using the `display` command in Databricks.

```
stats_df = spark.read.csv("s3://dsp-ch6/csv/game_skater_stats.csv",
                           header=True, inferSchema = True)
stats_df.createOrReplaceTempView("stats")

new_df = spark.sql("""
    select player_id, sum(1) as games, sum(goals) as goals
    from stats
    group by 1
    order by 3 desc
    limit 5
""")

display(new_df)
```

An output of this code block is shown in Figure 6.4. It shows the highest scoring players in the NHL dataset by ranking the results based on the total number of goals. One of the powerful features of Spark is that the SQL query will not operate against the dataframe until a result set is needed. This means that commands in a notebook can set up multiple data transformation steps for Spark dataframes, which are not performed until a later step needs to execute the graph of operations defined in a code block.

Spark SQL is expressive, fast, and my go-to method for working with big data sets in a Spark environment. While prior Spark ver-

sions performed better with the Dataframe API versus Spark SQL, the difference in performance is now trivial and you should use the transformation tools that provide the best iteration speed for working with large data sets. With Spark SQL, you can join dataframes, run nested queries, set up temp tables, and mix expressive Spark operations with SQL operations. For example, if you want to look at the distribution of goals versus shots in the NHL stats data, you can run the following command on the dataframe.

```
display(spark.sql("""
    select cast(goals/shots * 50 as int)/50.0 as Goals_per_shot
           ,sum(1) as Players
    from (
        select player_id, sum(shots) as shots, sum(goals) as goals
        from stats
        group by 1
        having goals >= 5
    )
    group by 1
    order by 1
    """))
```

This query restricts the ratio of goals to shots to players with more than 5 goals, to prevent outliers such as goalies scoring during power plays. We'll use the `display` command to output the result set as a table and then use Databricks to display the output as a chart. Many Spark ecosystems have ways of visualizing results, and the Databricks environment provides this capability through the `display` command, which works well with both tabular and pivot table data. After running the above command, you can click on the chart icon and choose dimensions and measures which show the distribution of goals versus shots, as visualized in Figure 6.5.

While I'm an advocate of using SQL to transform data, since it scales to different programming environments, it's useful to get familiar with some of the basic dataframe operations in PySpark. The code snippet below shows how to perform common operations

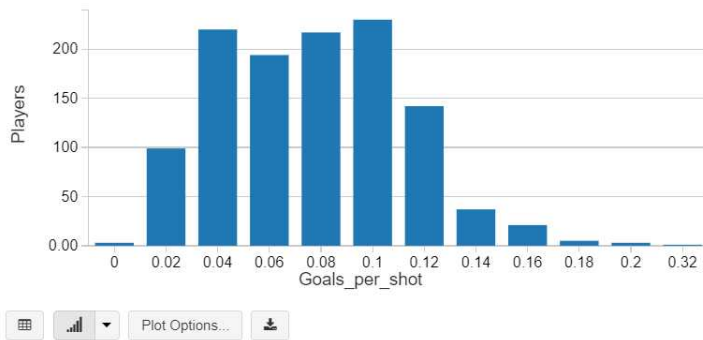


FIGURE 6.5: Distribution of goals per shot.

including dropping columns, selecting a subset of columns, and adding new columns to a Spark dataframe. Like prior commands, all of these operations are lazily performed. There are some syntax differences from Pandas, but the general commands used for transforming data sets should be familiar.

```
from pyspark.sql.functions import lit

# dropping columns
copy_df = stats_df.drop('game_id', 'player_id')

# selection columns
copy_df = copy_df.select('assists', 'goals', 'shots')

# adding columns
copy_df = copy_df.withColumn("league", lit('NHL'))
display(copy_df)
```

One of the common operations to perform between dataframes is a join. This is easy to express in a Spark SQL query, but sometimes it is preferable to do this programmatically with the Dataframe API. The code snippet below shows how to join two dataframes together, when joining on the `game_id` and `player_id` fields. The `league` column which is a literal will be joined with the rest of the stats dataframe. This is a trivial example where we are adding a

game_id	player_id	league	team_id	timeOnIce	assists	goals	shots	hits	powerPlayGoals	powerPlayAssists
2011030221	8467412	NHL	1	999	0	0	1	3	0	0
2011030221	8468501	NHL	1	1168	0	0	0	4	0	0
2011030221	8470609	NHL	1	558	0	0	2	1	0	0
2011030221	8471816	NHL	1	1134	0	0	1	4	0	0
2011030221	8472410	NHL	1	436	0	0	1	3	0	0
2011030221	8471233	NHL	1	1344	0	1	4	0	1	0

FIGURE 6.6: The dataframe resulting from the join.

new column onto a small dataframe, but the join operation from the Dataframe API can scale to massive data sets.

```
copy_df = stats_df.select('game_id', 'player_id').  
                  withColumn("league", lit('NHL'))  
df = copy_df.join(stats_df, ['game_id', 'player_id'])  
display(df)
```

The result set from the join operation above is shown in Figure 6.6. Spark supports a variety of different join types, and in this example we used an inner join to append the league column to the players stats dataframe.

It's also possible to perform aggregation operations on a dataframe, such as calculating sums and averages of columns. An example of computing the average time on ice for players in the stats data set, and total number of goals scored is shown in the snippet below. The `groupBy` command uses the `player_id` as the column for collapsing the data set, and the `agg` command specifies the aggregations to perform.

```
summary_df = stats_df.groupBy("player_id").agg(  
    {'timeOnIce': 'avg', 'goals': 'sum'})  
display(summary_df)
```

The snippet creates a dataframe with `player_id`, `timeOnIce`, and `goals` columns. We'll again use the plotting functionality in Databricks to visualize the results, but this time select the scatter

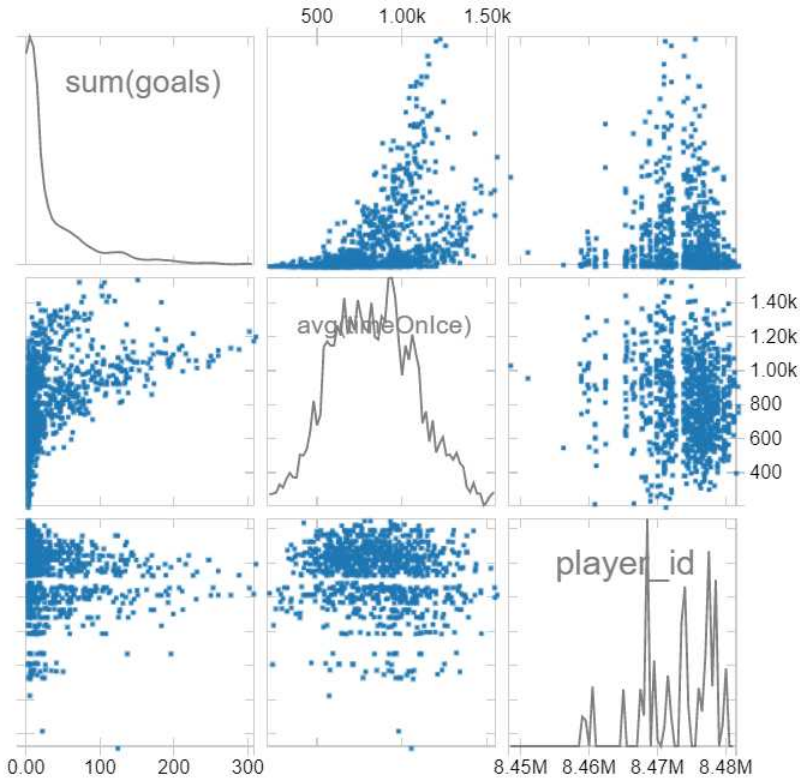


FIGURE 6.7: Time on ice and goal scoring plots.

plot option. The resulting plot of goals versus time on ice is shown in Figure 6.7.

We’ve worked through introductory examples to get up and running with dataframes in PySpark, focusing on operations that are useful for munging data prior to training machine learning models. These types of operations, in combination with reading and writing dataframes provide a useful set of skills for performing exploratory analysis on massive data sets.

6.3.4 Pandas UDFs

While PySpark provides a great deal of functionality for working with dataframes, it often lacks core functionality provided in Python libraries, such as the curve-fitting functions in SciPy.

While it's possible to use the `toPandas` function to convert dataframes into the Pandas format for Python libraries, this approach breaks when using large data sets. Pandas UDFs are a newer feature in PySpark that help data scientists work around this problem, by distributing the Pandas conversion across the worker nodes in a Spark cluster. With a Pandas UDF, you define a group by operation to partition the data set into dataframes that are small enough to fit into the memory of worker nodes, and then author a function that takes in a Pandas dataframe as the input parameter and returns a transformed Pandas dataframe as the result. Behind the scenes, PySpark uses the PyArrow library to efficiently translate dataframes from Spark to Pandas and back from Pandas to Spark. This approach enables Python libraries, such as Keras, to be scaled up to a large cluster of machines.

This section will walk through an example problem where we need to use an existing Python library and show how to translate the workflow into a scalable solution using Pandas UDFs. The question we are looking to answer is understanding if there is a positive or negative relationship between the `shots` and `hits` attributes in the `stats` data set. To calculate this relationship, we can use the `leastsq` function in SciPy, as shown in the snippet below. This example creates a Pandas dataframe for a single `player_id`, and then fits a simple linear regression between these attributes. The output is the coefficients used to fit the least-squares operation, and in this case the number of shots was not strongly correlated with the number of hits.

```
sample_pd = spark.sql("""
    select * from stats
    where player_id = 8471214
""").toPandas()

# Import Python libraries
from scipy.optimize import leastsq
import numpy as np
```

```
# Define a function to fit
def fit(params, x, y):
    return (y - (params[0] + x * params[1] ))

# Fit the curve and show the results
result = leastsq(fit, [1,0], args=(sample_pd.shots,sample_pd.hits))
print(result)
```

Now we want to perform this operation for every player in the stats data set. To scale to this volume, we'll first partition by `player_id`, as shown by the `groupBy` operation in the code snippet below. Next, We'll run the `analyze_player` function for each of these partitioned data sets using the `apply` command. While the `stats_df` dataframe used as input to this operation and the `players_df` dataframe returned are Spark dataframes, the `sampled_pd` dataframe and the dataframe returned by the `analyze_player` function are Pandas. The Pandas UDF annotation provides a hint to PySpark for how to distribute this workload so that it can scale the operation across the cluster of worker nodes rather than eagerly pulling all of the data to the driver node. Like most Spark operations, Pandas UDFs are lazily evaluated and will not be executed until an output value is needed.

Our initial example now translated to use Pandas UDFs is shown below. After defining additional modules to include, we specify the schema of the dataframe that will be returned from the operation. The `schema` object defines the structure of the Spark dataframe that will be returned from applying the `analyze_player` function. The next step in the code block lists an annotation that defines this function as a grouped map operation, which means that it works on dataframes rather than scalar values. As before, we'll use the `leastsq` function to fit the shots and hits attributes. After calculating the coefficients for this curve fitting, we create a new Pandas dataframe with the player id, and regression coefficients. The `display` command at the end of this code block will force the Pandas UDF to execute, which will create a partition for each of

the players in the data set, apply the least squares operation, and merge the results back together into a Spark dataframe.

```
# Load necessary libraries
from pyspark.sql.functions import pandas_udf, PandasUDFType
from pyspark.sql.types import *
import pandas as pd

# Create the schema for the resulting dataframe
schema = StructType([StructField('ID', LongType(), True),
                     StructField('p0', DoubleType(), True),
                     StructField('p1', DoubleType(), True)])

# Define the UDF, input and outputs are Pandas DFs
@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def analyze_player(sample_pd):

    # return empty params in not enough data
    if (len(sample_pd.shots) <= 1):
        return pd.DataFrame({'ID': [sample_pd.player_id[0]],
                             'p0': [ 0 ], 'p1': [ 0 ]})

    # Perform curve fitting
    result = leastsq(fit, [1, 0], args=(sample_pd.shots,
                                       sample_pd.hits))

    # Return the parameters as a Pandas DF
    return pd.DataFrame({'ID': [sample_pd.player_id[0]],
                        'p0': [result[0][0]], 'p1': [result[0][1]]})

# perform the UDF and show the results
player_df = stats_df.groupby('player_id').apply(analyze_player)
display(player_df)
```

The key capability that Pandas UDFs provide is that they enable Python libraries to be used in a distributed environment, as long as you have a good way of partitioning your data. This means that

ID	p0	p1
8470085	2.344963791971333	-0.15734035549738007
8471859	0.552176162823148	0.02217367140041992
8475765	0.7783287624631094	-0.00016742139167102827
8476426	-2.1813661987835076e-12	1.6666666666703023
8476439	2.2017087251697496	0.08646011684051094
8476445	0.166666666666484875	0.1666666666670303

FIGURE 6.8: The output dataframe from the Pandas UDF.

libraries such as Featuretools, which were not initially designed to work in a distributed environment, can be scaled to a large cluster. The result of applying the Pandas UDF from above on the stats data set is shown in Figure 6.8. This feature enables a mostly seamless translation between different dataframe formats.

To further demonstrate the value of Pandas UDFs, we'll apply them to distributing a feature generation pipeline and a deep learning pipeline. However, there are some issues when using Pandas UDFs in workflows, because they can make debugging more of a challenge and sometimes fail due to data type mismatches between Spark and Pandas.

6.3.5 Best Practices

While PySpark provides a familiar environment for Python programmers, it's good to follow a few best practices to make sure you are using Spark efficiently. Here are a set of recommendations I've compiled based on my experience porting a few projects from Python to PySpark:

- **Avoid dictionaries:** Using Python data types such as dictionaries means that the code might not be executable in a distributed mode. Instead of using keys to index values in a dictionary, consider adding another column to a dataframe that can be used as a filter. This recommendation applies to other Python types including lists that are not distributable in PySpark.
- **Limit Pandas usage:** Calling `toPandas` will cause all data to be loaded into memory on the driver node, and prevents operations from being performed in a distributed mode. It's fine to use this

function when data has already been aggregated and you want to make use of familiar Python plotting tools, but it should not be used for large dataframes.

- **Avoid loops:** Instead of using for loops, it's often possible to use functional approaches such as group by and apply to achieve the same result. Using this pattern means that code can be parallelized by supported execution environments. I've noticed that focusing on using this pattern in Python has also resulted in cleaner code that is easier to translate to PySpark.
- **Minimize eager operations:** In order for your pipeline to be as scalable as possible, it's good to avoid eager operations that pull full dataframes into memory. For example, reading in CSVs is an eager operation, and my work around is to stage the dataframe to S3 as Parquet before using it in later pipeline steps.
- **Use SQL:** There are libraries that provide SQL operations against dataframes in both Python and PySpark. If you're working with someone else's Python code, it can be tricky to decipher what some of the Pandas operations are achieving. If you plan on porting your code from Python to PySpark, then using a SQL library for Pandas can make this translation easier.

By following these best practices when writing PySpark code, I've been able to improve both my Python and PySpark data science workflows.

6.4 MLlib Batch Pipeline

Now that we've covered loading and transforming data with PySpark, we can now use the machine learning libraries in PySpark to build a predictive model. The core library for building predictive models in PySpark is called MLlib. This library provides a suite of supervised and unsupervised algorithms. While this library does not have complete coverage of all of the algorithms in sklearn, it provides functionality for the majority of the types of operations

needed for data science workflows. In this chapter, we'll show how to apply MLlib to a classification problem and save the outputs from the model application to a data lake.

```
games_df = spark.read.csv("s3://dsp-ch6/csv/games-expand.csv",
                           header=True, inferSchema = True)
games_df.createOrReplaceTempView("games_df")

games_df = spark.sql("""
    select *, row_number() over (order by rand()) as user_id
    ,case when rand() > 0.7 then 1 else 0 end as test
    from games_df
""")
```

The first step in the pipeline is loading the data set that we want to use for model training. The snippet above shows how to load the games data set, and append two additional attributes to the loaded dataframe using Spark SQL. The result of running this query is that about 30% of users will be assigned a test value which we'll use for model application, and each record is assigned a unique user ID which we'll use when saving the model predictions.

The next step is splitting up the data set into train and test dataframes. For this pipeline, we'll use the test dataframe as the data set for model application, where we predict user behavior. An example of splitting up the dataframes using the test column is shown in the snippet below. This should result in roughly 16.1k training users and 6.8k test users.

```
trainDF = games_df.filter("test == 0")
testDF = games_df.filter("test == 1")
print("Train " + str(trainDF.count()))
print("Test " + str(testDF.count()))
```

6.4.1 Vector Columns

MLlib requires that the input data is formatted using vector data types in Spark. To transform our dataframe into this format, we can use the `VectorAssembler` class to combine a range of columns into a single vector column. The code snippet below shows how to use this class to merge the first 10 columns in the dataframe into a new vector column called `features`. After applying this command to the training dataframe using the `transform` function, we use the `select` function to retrieve only the values we need from the dataframe for model training and application. For the training dataframe, we only need the label and features, and with the test dataframe we also select the user ID.

```
from pyspark.ml.feature import VectorAssembler

# create a vector representation
assembler = VectorAssembler(
    inputCols= trainDF.schema.names[0:10],
    outputCol="features" )

trainVec = assembler.transform(trainDF).select('label', 'features')
testVec = assembler.transform(testDF).select(
    'label', 'features', 'user_id')
display(testVec)
```

The `display` command shows the result of transforming our test dataset into vector types usable by MLlib. The output dataframe is visualized in Figure 6.9.

6.4.2 Model Application

Now that we have prepared our training and test data sets, we can use the logistic regression algorithm provided by MLlib to fit the training dataframe. We first create a logistic regression object and define the columns to use as labels and features. Next, we use the `fit` function to train the model on the training data set. In the last

label	features	user_id
1	[0, 10, [0], [1]]	1
0	[0, 10, [0, 2], [1, 1]]	3
0	[0, 10, [], []]	7
0	[0, 10, [2], [1]]	14
0	[0, 10, [2], [1]]	17
0	[0, 10, [7], [1]]	18
0	[0, 10, [1], [1]]	21
0	[0, 10, [], []]	22
0	[0, 10, [7], [1]]	24

FIGURE 6.9: The features in the sparse vector format.

step in the snippet below, we use the `transform` function to apply the model to our test data set.

```
from pyspark.ml.classification import LogisticRegression

# specify the columns for the model
lr = LogisticRegression(featuresCol='features', labelCol='label')

# fit on training data
model = lr.fit(trainVec)

# predict on test data
predDF = model.transform(testVec)
```

The resulting dataframe now has a `probability` column, as shown in Figure 6.10. This column is a 2-element array with the probabilities for class 0 and 1. To test the accuracy of the logistic regression model on the test data set, we can use the binary classification evaluator in Mlib to calculate the ROC metric, as shown in the snippet below. For my run of the model, the ROC metric has a value of 0.761.

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

roc = BinaryClassificationEvaluator().evaluate(predDF)
print(roc)
```


label	features	user_id	rawPrediction	probability	prediction	propensity
1	[0.10,[0],[1]]	1	[1.2,[2.6201307263149594,-2.6201307263149594]]	[1.2,[0.9321459753107263,0.06785402468927368]]	0	0.067854024
0	[0.10,[0.2],[1,1]]	3	[1.2,[2.2388549781523457,-2.2388549781523457]]	[1.2,[0.9036848433067928,0.09631515669320731]]	0	0.09631516
0	[0.10,[,][,]]	7	[1.2,[2.189921465856445,-2.189921465856445]]	[1.2,[0.899340797204121,0.10065920279587895]]	0	0.10065921
0	[0.10,[2],[1]]	14	[1.2,[1.8086457176938315,-1.8086457176938315]]	[1.2,[0.8591981173301758,0.1408018826698241]]	0	0.14080188
0	[0.10,[2],[1]]	17	[1.2,[1.8086457176938315,-1.8086457176938315]]	[1.2,[0.8591981173301758,0.1408018826698241]]	0	0.14080188

FIGURE 6.10: The dataframe with propensity scores.

In a production pipeline, there will not be labels for the users that need predictions, meaning that you'll need to perform cross validation to select the best model for making predictions. An example of this approach is covered in Section 6.7. In this case, we are using a single data set to keep code examples short, but a similar pipeline can be used in a production workflows.

Now that we have the model predictions for our test users, we need to retrieve the predicted label in order to create a dataframe to persist to a data lake. Since the probability column created by MLlib is an array, we'll need to define a UDF that retrieves the second element as our propensity column, as shown in the PySpark snippet below.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType

# split out the array into a column
secondElement = udf(lambda v:float(v[1]),FloatType())
predDF = predDF.select("*",
                        secondElement("probability").alias("propensity"))
display(predDF)
```

After running this code block, the dataframe will have an additional column called `propensity` as shown in Figure 6.10. The final step in this batch prediction pipeline is to save the results to S3. We'll use the `select` function to retrieve the relevant columns from the predictions dataframe, and then use the `write` function on the dataframe to persist the results as Parquet on S3.

```
# save results to S3
results_df = predDF.select("user_id", "propensity")
results_path = "s3a://dsp-ch6/game-predictions/"
results_df.write.mode('overwrite').parquet(results_path)
```

We now have all of the building blocks needed to create a PySpark pipeline that can fetch data from a storage layer, train a predictive model, and write the results to persistent storage. We'll cover how to schedule this type of pipeline in Section 6.8.

When developing models, it's useful to inspect the output to see if the distribution of model predictions matches expectations. We can use Spark SQL to perform an aggregation on the model outputs and then use the display command to perform this process directly in Databricks, as shown in the snippet below. The result of performing these steps on the model predictions is shown in Figure 6.11.

```
# plot the predictions
predDF .createOrReplaceTempView("predDF ")

plotDF = spark.sql("""
    select cast(propensity*100 as int)/100 as propensity,
           label, sum(1) as users
    from predDF
    group by 1, 2
    order by 1, 2
""")

# table output
display(plotDF)
```

MLlib can be applied to a wide variety of problems using a large suite of algorithms. While we explored logistic regression in this section, the libraries provides a number of different classification

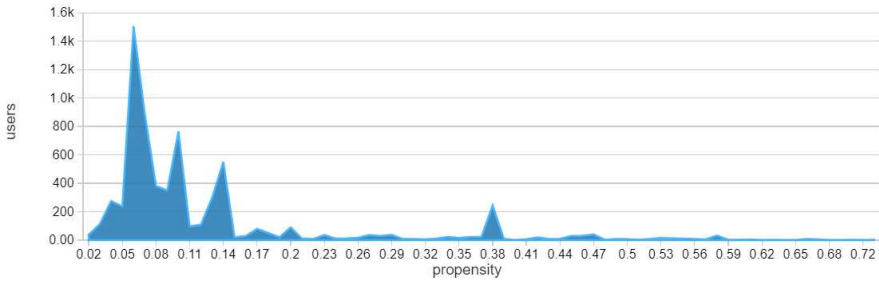


FIGURE 6.11: The distribution of propensity scores.

approaches, and there are other types of operations supported including regression and clustering.

6.5 Distributed Deep Learning

While MLlib provides scalable implementations for classic machine learning algorithms, it does not natively support deep learning libraries such as TensorFlow and PyTorch. There are libraries that parallelize the training of deep learning models on Spark, but the data set needs to be able to fit in memory on each worker node, and these approaches are best used for distributed hyperparameter tuning on medium-sized data sets.

For the model application stage, where we already have a deep learning model trained but need to apply the resulting model to a large user base, we can use Pandas UDFs. With Pandas UDFs, we can partition and distribute our data set, run the resulting dataframes against a Keras model, and then compile the results back into a single large Spark dataframe. This section will show how we can take the Keras model that we built in Section 1.6.3, and scale it to larger data sets using PySpark and Pandas UDFs. However, we still have the requirement that the data used for training the model can fit into memory on the driver node.

We'll use the same data sets from the prior section, where we split the games data set into training and test sets of users. This is a

relatively small data set, so we can use the `toPandas` operation to load the dataframe onto the driver node, as shown in the snippet below. The result is a dataframe and list that we can provide as input to train a Keras deep learning model.

```
# build model on the driver node
train_pd = trainDF.toPandas()
x_train = train_pd.iloc[:,0:10]
y_train = train_pd['label']
```

When using PyPI to install TensorFlow on the Spark cluster, the installed version of the library should be 2.0 or greater. This differs from Version 1 of TensorFlow, which we used in prior chapters. The main impact in terms of the code snippets is that TensorFlow 2 now has a built-in AUC function that no longer requires the workflow we previously applied.

6.5.1 Model Training

We'll use the same approach as before to train a Keras model. The code snippet below shows how to set up a network with an input layer, dropout later, single hidden layer, and an output layer, optimized with `rmsprop` and a cross entropy loss. In the model application phase, we'll reuse the `model` object in a Pandas UDFs to distribute the workload.

```
import tensorflow as tf
import keras
from keras import models, layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10,)))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy')
```

```
history = model.fit(x_train, y_train, epochs=100, batch_size=100,
                    validation_split = .2, verbose=0)
```

To test for overfitting, we can plot the results of the training and validation data sets, as shown in Figure 6.12. The snippet below shows how to use matplotlib to display the losses over time for these data sets. While the training loss continued to decrease over additional epochs, the validation loss stopped improving after 20 epochs, but did not noticeably increase over time.

```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)

fig = plt.figure(figsize=(10,6) )
plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'b', label='Validation Loss')
plt.legend()
plt.show()
display(fig)
```

6.5.2 Model Application

Now that we have a trained deep learning model, we can use PySpark to apply it in a scalable pipeline. The first step is determining how to partition the set of users that need to be scored. For this data set, we can split the user base into 100 different buckets, as shown in the snippet below. This randomly assigns each user into 1 of 100 buckets, which means that after applying the group by step, each dataframe that gets translated to Pandas will be roughly 1% of the size of the original dataframe. If you have a large data set, you may need to use thousands of buckets to distribute the data set, and maybe more.

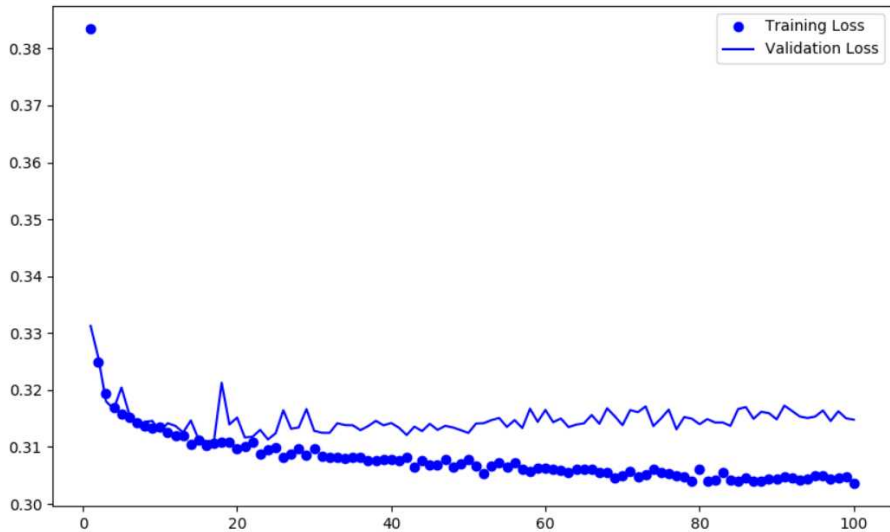


FIGURE 6.12: Training a Keras model on a subset of data.

```
# set up partitioning for the train dataframe
testDF.createOrReplaceTempView("testDF ")

partitionedDF = spark.sql("""
    select *, cast(rand()*100 as int) as partition_id
    from testDF
""")
```

The next step is to define the Pandas UDF that will apply the Keras model. We'll define an output schema of a user ID and propensity score, as shown below. The UDF uses the `predict` function on the model object we previously trained to create a `prediction` column on the passed in dataframe. The return command selects the two relevant columns that we defined for the schema object. The group by command partitions the data set using our bucketing approach, and the apply command performs the Keras model application across the cluster of worker nodes. The result is a Spark dataframe visualized with the display command, as shown in Figure 6.13.

user_id	propensity
111	0.12130042910575867
338	0.018362104892730713
352	0.016818374395370483
657	0.08839917182922363
2204	0.012326300144195557
2501	0.03954136371612549
2703	0.3526822328567505
3504	0.037293970584869385
4780	0.33306148648262024

FIGURE 6.13: The resulting dataframe for distributed Keras.

```
from pyspark.sql.functions import pandas_udf, PandasUDFType
from pyspark.sql.types import *

schema = StructType([StructField('user_id', LongType(), True),
                      StructField('propensity', DoubleType(), True)])

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def apply_keras(pd):
    pd['propensity'] = model.predict(pd.iloc[:,0:10])
    return pd[['user_id', 'propensity']]

results_df=partitionedDF.groupby('partition_id').apply(apply_keras)
display(results_df)
```

One thing to note is that there are limitations on the types of objects that you can reference in a Pandas UDFs. In this example, we referenced the `model` object, which was created on the driver node when training the model. When variables in PySpark are transferred from the driver node to workers nodes for distributed operations, a copy of the variable is made, because synchronizing variables across a cluster would be inefficient. This means that any changes made to a variable within a Pandas UDF will not apply to the original object. It's also why data types such as Python lists and dictionaries should be avoided when using UDFs. Functions work in a similar way, and in Section 6.3.4 we used the `fit` function

in a Pandas UDF where the function was initially defined on the driver node. Spark also provides broadcast variables for sharing variables in a cluster, but ideally distributed code segments should avoid sharing state through variables if possible.

6.6 Distributed Feature Engineering

Feature engineering is a key step in a data science workflow, and sometimes it is necessary to use Python libraries to implement this functionality. For example, the AutoModel system at Zynga uses the Featuretools library to generate hundreds of features from raw tracking events, which are then used as input to classification models. To scale up the automated feature engineering approach that we first explored in Section 1.7, we can use Pandas UDFs to distribute the feature application process. Like the prior section, we need to sample data when determining the transformation to perform, but when applying the transformation we can scale to massive data sets.

For this section, we'll use the game plays data set from the NHL Kaggle example, which includes detailed play-by-play descriptions of the events that occurred during each match. Our goal is to transform the deep and narrow dataframe into a shallow and wide dataframe that summarizes each game as a single record with hundreds of columns. An example of loading this data in PySpark and selecting the relevant columns is shown in the snippet below. Before calling `toPandas`, we use the `filter` function to sample 0.3% of the records, and then cast the result to a Pandas frame, which has a shape of 10,717 rows and 16 columns.

```
plays_df = spark.read.csv("s3://dsp-ch6/csv/game_plays.csv",
                          header=True, inferSchema = True).drop(
                              'secondaryType', 'periodType', 'dateTime', 'rink_side')
plays_pd = plays_df.filter("rand() < 0.003").toPandas()
plays_pd.shape
```


6.6.1 Feature Generation

We'll use the same two-step process covered in Section 1.7 where we first one-hot encode the categorical features in the dataframe, and then apply deep feature synthesis to the data set. The code snippet below shows how to perform the encoding process using the Featuretools library. The output is a transformation of the initial dataframe that now has 20 dummy variables instead of the event and description variables.

```
import featuretools as ft
from featuretools import Feature

es = ft.EntitySet(id="plays")
es = es.entity_from_dataframe(entity_id="plays", dataframe=plays_pd,
                             index="play_id", variable_types = {
                                 "event": ft.variable_types.Categorical,
                                 "description": ft.variable_types.Categorical })

f1 = Feature(es["plays"]["event"])
f2 = Feature(es["plays"]["description"])

encoded, defs = ft.encode_features(plays_pd, [f1, f2], top_n=10)
encoded.reset_index(inplace=True)
```

The next step is using the `dfs` function to perform deep feature synthesis on our encoded dataframe. The input dataframe will have a record per play, while the output dataframe will have a single record per game after collapsing the detailed events into a wide column representation using a variety of different aggregations.

```
es = ft.EntitySet(id="plays")
es = es.entity_from_dataframe(entity_id="plays",
                             dataframe=encoded, index="play_id")

es = es.normalize_entity(base_entity_id="plays",
                        new_entity_id="games", index="game_id")
```

```
Out[56]: Index(['game_id', 'SUMplaysdescriptionIcing', 'SUMplayseventPenalty',
               'SUMplayseventTakeaway', 'SUMplaysgoals_away',
               'SUMplaysdescriptionPeriodOfficial', 'SUMplayseventShot',
               'SUMplaysdescriptionPuckinBenches', 'SUMplaysdescriptionPuckinNetting',
               'SUMplaysdescriptionPeriodReady',
               ...
               'NUM_UNIQUEplaysst_x', 'NUM_UNIQUEplaysx', 'NUM_UNIQUEplaysst_y',
               'NUM_UNIQUEplaysteam_id_against', 'MODEplaysteam_id_for', 'MODEplaysy',
               'MODEplaysst_x', 'MODEplaysx', 'MODEplaysst_y',
               'MODEplaysteam_id_against'],
              dtype='object', length=188)
```

FIGURE 6.14: The schema for the generated features.

```
features, transform=ft.dfs(entityset=es,
                           target_entity="games",max_depth=2)
features.reset_index(inplace=True)
```

One of the new steps that we need to perform versus the prior approach, is that we need to determine what the schema will be for the generated features, since this is needed as an input to the Pandas UDF annotation. To figure out what the generated schema is for the generated dataframe, we can create a Spark dataframe and then retrieve the schema from the dataframe. Before converting the Pandas dataframe, we need to modify the column names in the generated dataframe to remove special characters, as shown in the snippet below. The resulting Spark schema for the feature application step is displayed in Figure 6.14.

```
features.columns = features.columns.str.replace("[()\. =]", "")
schema = sqlContext.createDataFrame(features).schema
features.columns
```

We now have the required schema for defining a Pandas UDF. Unlike the past UDFs we defined, the schema may change between different runs based on the feature transformation aggregations selected by Featuretools. In these steps, we also created a `defs` object that defines the feature transformations to use for encoding and a `transform` object that defines the transformations to perform

deep feature synthesis. Like the model object in the past section, copies of these objects will be passed to the Pandas UDF executing on worker nodes.

6.6.2 Feature Application

To enable our approach to scale across a cluster of worker nodes, we need to define a column to use for partitioning. Like the prior section, we can bucket events into different sets of data to ensure that the UDF process can scale. One difference from before is that we need all of the plays from a specific game to be grouped into the same partition. To achieve this result, we can partition by the `game_id` rather than the `player_id`. An example of this approach is shown in the code snippet below. Additionally, we can use the hash function on the game ID to randomize the value, resulting in more balanced bucket sizes.

```
# bucket IDs
plays_df.createOrReplaceTempView("plays_df")
plays_df = spark.sql("""
    select *, abs(hash(game_id))%1000 as partition_id
    from plays_df
""")
```

We can now apply feature transformation to the full data set, using the Pandas UDF defined below. The plays dataframe is partitioned by the bucket before being passed to the generate features function. This function uses the previously generated feature transformations to ensure that the same transformation is applied across all of the worker nodes. The input Pandas dataframe is a narrow and deep representation of play data, while the returned dataframe is a shallow and wide representation of game summaries.

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def gen_features(plays_pd):
```

game_id	SUMplaysdescriptionIcing	SUMplaysevenPenalty	SUMplaysevenTakeaway	SUMplaysgoals_away	SUMplaysdescriptionPeriodOfficial	SUMplaysevenShot
2011030222	0	0	0	1	0	2
2011030223	0	1	0	2	0	0
2011030224	0	0	0	2	0	0
2011030324	0	0	0	1	0	1
2011030225	0	0	0	2	0	0
2011030232	0	0	0	5	0	0
2011030412	0	0	0	1	0	0

FIGURE 6.15: Generated features in the Spark dataframe.

```

es = ft.EntitySet(id="plays")
es = es.entity_from_dataframe(entity_id="plays",
                             dataframe=plays_pd, index="play_id", variable_types = {
                                 "event": ft.variable_types.Categorical,
                                 "description": ft.variable_types.Categorical })
encoded_features = ft.calculate_feature_matrix(defs, es)
encoded_features.reset_index(inplace=True)

es = ft.EntitySet(id="plays")
es = es.entity_from_dataframe(entity_id="plays",
                             dataframe=encoded, index="play_id")
es = es.normalize_entity(base_entity_id="plays",
                        new_entity_id="games", index="game_id")
generated = ft.calculate_feature_matrix(transform, es).fillna(0)

generated.reset_index(inplace=True)
generated.columns = generated.columns.str.replace("[()\. =]", "")
return generated

features_df = plays_df.groupby('partition_id').apply(gen_features)
display(features_df)

```

The output of the display command is shown in Figure 6.15. We've now worked through feature generation and deep learning in scalable model pipelines. Now that we have a transformed data set, we can join the result with additional features, such as the label that we are looking to predict, and develop a complete model pipeline.

6.7 GCP Model Pipeline

A common workflow for batch model pipelines is reading input data from a lake, applying a machine learning model, and then writing the results to an application database. In GCP, BigQuery serves as the data lake and Cloud Datastore can serve as an application database. We'll build an end-to-end pipeline with these components in the next chapter, but for now we'll get hands on with a subset of the GCP components directly in Spark.

While there is a Spark connector for BigQuery³, enabling large-scale PySpark pipelines to be built using BigQuery directly, there are some issues with this library that make it quite complicated to set up for our Databricks environment. For example, we would need to rebuild some of the JAR files and shade the dependencies. One alternative is to use the Python BigQuery connector that we explored in Section 5.1, but this approach is not distributed and will eagerly pull the query results to the driver node as a Pandas dataframe. For this chapter, we'll explore a workflow where we unload query results to Cloud Storage, and then read in the data set from GCS as the initial step in the pipeline. Similarly, for model output we'll save the results to GCS, where the output is available for pushing to Cloud Datastore. To productize this type of model workflow, Airflow could be used to chain these different actions together.

6.7.1 BigQuery Export

The first step we'll perform is exporting the results of a BigQuery query to GCS, which can be performed manually using the BigQuery UI. This is possible to perform directly in Spark, but as I mentioned the setup is quite involved to configure with the current version of the connector library. We'll use the `natality` data set for this pipeline, which lists attributes about child deliveries, such as birth weight.

³<https://github.com/spotify/spark-bigquery/>

```
create table dsp_demo.natality as (  
  select *  
  from `bigquery-public-data.samples.natality`  
  order by rand()  
  limit 10000  
)
```

To create a data set, we'll sample 10k records from the natality public data set in BigQuery. To export this result set to GCS, we need to create a table on BigQuery with the data that we want to export. The SQL for creating this data sample is shown in the snippet above. To export this data to GCS, perform the following steps:

1. Browse to the GCP Console
2. Search for “BigQuery”
3. Paste the Query from the snippet above into the editor
4. Click Run
5. In the left pane, select the table, “dsp_demo.natality”
6. Click “Export”, and then “Export to GCS”
7. Set the location, “/dsp_model_store/natality/avro”
8. Use “Avro” as export format
9. Click “Export”

After performing these steps, the sampled natality data will be saved to GCS in Avro format. The confirmation dialog from exporting the data set is shown in Figure 6.16. We now have the data saved to GCS in a format that works well with Spark.

6.7.2 GCP Credentials

We now have a data set that we can use as input to a PySpark pipeline, but we don't yet have access to the bucket on GCS from our Spark environment. With AWS, we were able to set up programmatic access to S3 using an access and secret key. With GCP, the process is a bit more complicated because we need to move the json credentials file to the driver node of the cluster in order to

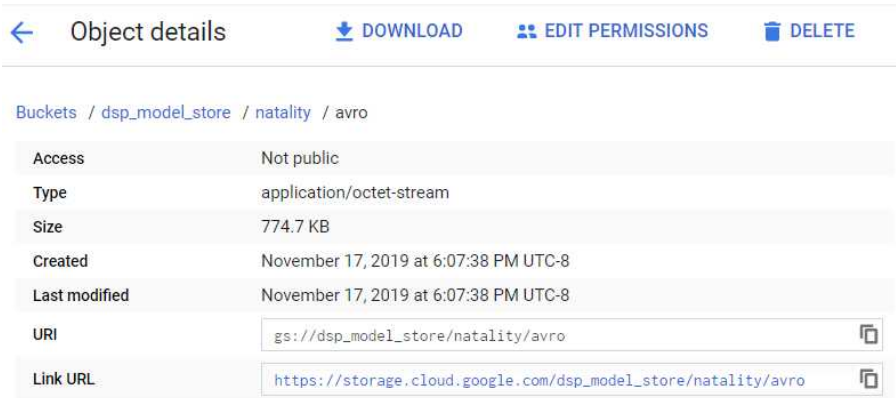


FIGURE 6.16: Confirming the Avro export on GCS.

read and write files on GCS. One of the challenges with using Spark is that you may not have SSH access to the driver node, which means that we'll need to use persistent storage to move the file to the driver machine. This isn't recommended for production environments, but instead is being shown as a proof of concept. The best practice for managing credentials in a production environment is to use IAM roles.

```
aws s3 cp dsdemo.json s3://dsp-ch6/secrets/dsdemo.json

aws s3 ls s3://dsp-ch6/secrets/
```

To move the json file to the driver node, we can first copy the credentials file to S3, as shown in the snippet above. Now we can switch back to Databricks and author the model pipeline. To copy the file to the driver node, we can read in the file using the `sc` Spark context to read the file line by line. This is different from all of our prior operations where we have read in data sets as dataframes. After reading the file, we then create a file on the driver node using the Python `open` and `write` functions. Again, this is an unusual action to perform in Spark, because you typically want to write to persistent storage rather than local storage. The

result of performing these steps is that the credentials file will now be available locally on the driver node in the cluster.

```
creds_file = '/databricks/creds.json'
creds = sc.textFile('s3://dsp-ch6/secrets/dsdemo.json')

with open(creds_file, 'w') as file:
    for line in creds.take(100):
        file.write(line + "\n")
```

Now that we have the json credentials file moved to the driver local storage, we can set up the Hadoop configuration needed to access data on GCS. The code snippet below shows how to configure the project ID, file system implementation, and credentials file location. After running these commands, we now have access to read and write files on GCS.

```
sc._jsc.hadoopConfiguration().set("fs.gs.impl",
    "com.google.cloud.hadoop.fs.gcs.GoogleHadoopFileSystem")
sc._jsc.hadoopConfiguration().set("fs.gs.project.id",
    "your_project_id")

sc._jsc.hadoopConfiguration().set(
    "mapred.bq.auth.service.account.json.keyfile", creds_file)
sc._jsc.hadoopConfiguration().set(
    "fs.gs.auth.service.account.json.keyfile", creds_file)
```

6.7.3 Model Pipeline

To read in the natality data set, we can use the read function with the Avro setting to fetch the data set. Since we are using the Avro format, the dataframe will be lazily loaded and the data is not retrieved until the display command is used to sample the data set, as shown in the snippet below.


```
natality_path = "gs://dsp_model_store/natality/avro"
natality_df = spark.read.format("avro").load(natality_path)
display(natality_df)
```

Before we can use MLlib to build a regression model, we need to perform a few transformations on the data set to select a subset of the features, cast data types, and split records into training and test groups. We'll also use the `fillna` function as shown below in order to replace any null values in the dataframe with zeros. For this modeling exercise, we'll build a regression model that predicts the birth weight of a baby using a few different features including the marriage status of the mother and parent ages. The prepared dataframe is shown in Figure 6.17.

```
natality_df.createOrReplaceTempView("natality_df")

natality_df = spark.sql("""
SELECT year, plurality, apgar_5min,
       mother_age, father_age,
       gestation_weeks, ever_born
       ,case when mother_married = true
             then 1 else 0 end as mother_married
       ,weight_pounds as weight
       ,case when rand() < 0.5 then 1 else 0 end as test
from natality_df
""").fillna(0)

trainDF = natality_df.filter("test == 0")
testDF = natality_df.filter("test == 1")
display(natality_df)
```

Next, we'll translate our dataframe into the vector data types that MLlib requires as input. The process for transforming the natality data set is shown in the snippet below. After executing the `transform` function, we now have training and test data sets we can use

year	plurality	apgar_5min	mother_age	father_age	gestation_weeks	ever_born	mother_married	weight	test
1984	1	99	25	29	41	1	1	8.12623897732	0
2006	1	9	22	99	41	2	0	9.56365292556	1
1985	1	9	19	99	41	1	0	6.9996768185	0
1998	1	9	19	21	36	1	1	5.8135898489399995	1
1982	1	9	28	28	41	2	1	8.56275425608	0
1991	1	9	17	20	35	1	0	8.50102482272	0
1980	1	9	21	23	35	2	1	5.00008410216	1
1997	1	9	23	26	39	1	1	8.56275425608	0
1974	1	0	17	21	42	1	0	8.437090766739999	0

FIGURE 6.17: The prepared Natality dataframe.

as input to a regression model. The label we are building a model to predict is the `weight` column.

```
from pyspark.ml.feature import VectorAssembler

# create a vector representation
assembler = VectorAssembler(inputCols= trainDF.schema.names[0:8],
                             outputCol="features" )

trainVec = assembler.transform(trainDF).select('weight','features')
testVec = assembler.transform(testDF).select('weight', 'features')
```

MLlib provides a set of utilities for performing cross validation and hyperparameter tuning in a model workflow. The code snippet below shows how to perform this process for a random forest regression model. Instead of calling `fit` directly on the model object, we wrap the model object with a cross validator object that explores different parameter settings, such as tree depth and number of trees. This workflow is similar to the grid search functions in `sklearn`. After searching through the parameter space, and using cross validation based on the number of folds, the random forest model is retrained on the complete training data set before being applied to make predictions on the test data set. The result is a dataframe with the actual weight and predicted birth weight.

```
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.tuning import CrossValidator
```

```
from pyspark.ml.evaluation import RegressionEvaluator

folds = 3
rf_trees = [ 50, 100 ]
rf_depth = [ 4, 5 ]

rf= RandomForestRegressor(featuresCol='features',labelCol='weight')

paramGrid = ParamGridBuilder().addGrid(rf.numTrees, rf_trees).
                                ddGrid(rf.maxDepth, rf_depth).build()
crossval = CrossValidator(estimator=rf, estimatorParamMaps =
                           paramGrid, evaluator=RegressionEvaluator(
                               labelCol='weight'), numFolds = folds)
rfModel = crossval.fit(trainVec)

predsDF = rfModel.transform(testVec).select("weight", "prediction")
```

In the final step of our GCP model pipeline, we'll save the results to GCS, so that other applications or processes in a workflow can make use of the predictions. The code snippet below shows how to write the dataframe to GCS in Avro format. To ensure that different runs of the pipeline do not overwrite past predictions, we append a timestamp to the export path.

```
import time

out_path = "gs://dsp_model_store/natality/preds-{time}"/".
            format(time = int(time.time()*1000))
predsDF.write.mode('overwrite').format("avro").save(out_path)
print(out_path)
```

Using GCP components with PySpark took a bit of effort to configure, but in this case we are running Spark in a different cloud provider than where we are reading and writing data. In a production environment, you'll most likely be running Spark in the same cloud as where you are working with data sets, which means

that you can leverage IAM roles for properly managing access to different services.

6.8 Productizing PySpark

Once you've tested a batch model pipeline in a notebook environment, there are a few different ways of scheduling the pipeline to run on a regular schedule. For example, you may want a churn prediction model for a mobile game to run every morning and publish the scores to an application database. Similar to the workflow tools we covered in Chapter 5, a PySpark pipeline should have monitoring in place for any failures that may occur. There's a few different approaches for scheduling PySpark jobs to run:

- **Workflow Tools:** Airflow, Azkaban, and Luigi all support running spark jobs as part of a workflow.
- **Cloud Tools:** EMR on AWS and Dataproc on GCP support scheduled Spark jobs.
- **Vendor Tools:** Databricks supports setting up job schedules with monitoring through the web UI.
- **Spark Submit:** If you already have a cluster provisioned, you can issue `spark-submit` commands using a tool such as `crontab`.

Vendor and cloud tools are typically easier to get up and running, because they provide options for provisioning clusters as part of the workflow. For example, with Databricks you can define the type of cluster to spin up for running a notebook on a schedule. When using a workflow tool, such as Airflow, you'll need to add additional steps to your workflow in order to spin up and terminate clusters. Most workflow tools provide connectors to EMR for managing clusters as part of a workflow. The Spark submit option is useful when first getting started with scheduling Spark jobs, but it doesn't support managing clusters as part of a workflow.

Spark jobs can run on ephemeral or persistent clusters. An ephemeral cluster is a Spark cluster that is provisioned to perform a set of tasks and then terminated, such as running a churn

model pipeline. A persistent cluster is a long-running cluster than may support interactive notebooks, such as the Databricks cluster we set up at the start of this chapter. Persistent clusters are useful for development, but can be expensive if the hardware spun up for the cluster is under utilized. Some vendors support auto scaling of clusters to reduce the cost of long-running persistent clusters. Ephemeral clusters are useful, because spinning up a new cluster to perform a task enables isolation of failure across tasks, and it means that different model pipelines can use different library versions and Spark runtimes.

In addition to setting up tools for scheduling jobs and alerting on job failures, it's useful to set up additional data and model quality checks for Spark model pipelines. For example, I've set up Spark jobs that perform audit tasks, such as making sure that an application database has predictions for the current day, and trigger alerts if prediction data is stale. It's also a good practice to log metrics, such as the ROC of a cross-validated model, as part of a Spark pipeline.

6.9 Conclusion

PySpark is a powerful tool for data scientists to build scalable analyses and model pipelines. It a highly desirable skill set for companies, because it enables data science teams to own more of the process of building and owning data products. There's a variety of ways to set up an environment for PySpark, and in this chapter we explored a free notebook environment from one of the popular Spark vendors.

This chapter focused on batch model pipelines, where the goal is to create a set of predictions for a large number of users on a regular schedule. We explored pipelines for both AWS and GCP deployments, where the data sources and data outputs are data lakes. One of the issues with these types of pipelines is that predictions may be quite stale by the time that a prediction is used. In

Chapter 8, we'll explore streaming pipelines for PySpark, where the latency of model predictions is minimized.

PySpark is a highly expressive language for authoring model pipelines, because it supports all Python functionality, but does require some workarounds to get code to execute across a cluster of workers nodes. In the next chapter we'll explore Dataflow, a runtime for the Apache Beam library, which also enables large-scale distributed Python pipelines, but is more constrained in the types of operations that you can perform.

Cloud Dataflow for Batch Modeling

Dataflow is a tool for building data pipelines that can run locally, or scale up to large clusters in a managed environment. While Cloud Dataflow was initially incubated at Google as a GCP specific tool, it now builds upon the open-source Apache Beam library, making it usable in other cloud environments. The tool provides input connectors to different data sources, such as BigQuery and files on Cloud Storage, operators for transforming and aggregating data, and output connectors to systems such as Cloud Datastore and BigQuery.

In this chapter, we'll build a pipeline with Dataflow that reads in data from BigQuery, applies a sklearn model to create predictions, and then writes the predictions to BigQuery and Cloud Datastore. We'll start by running the pipeline locally on a subset of data and then scale up to a larger data set using GCP.

Dataflow is designed to enable highly-scalable data pipelines, such as performing ETL work where you need to move data between different systems in your cloud deployment. It's also been extended to work well for building ML pipelines, and there's built-in support for TensorFlow and other machine learning methods. The result is that Dataflow enables data scientists to build large scale pipelines without needing the support of an engineering team to scale things up for production.

The core component in Dataflow is a pipeline, which defines the operations to perform as part of a workflow. A workflow in Dataflow is a DAG that includes data sources, data sinks, and data transformations. Here are some of the key components:

- **Pipeline:** Defines the set of operations to perform as part of a Dataflow job.
- **Collection:** The interface between different stages in a workflow. The input to any step in a workflow is a collection of objects and the output is a new collection of objects.
- **DoFn:** An operation to perform on each element in a collection, resulting in a new collection.
- **Transform:** An operation to perform on sets of elements in a collection, such as an aggregation.

Dataflow works with multiple languages, but we'll focus on the Python implementation for this book. There are some caveats with the Python version, because worker nodes may need to compile libraries from source, but it does provide a good introduction to the different components in Apache Beam. To create a workflow with Beam, you use the pipe syntax in Python to chain different steps together. The result is a DAG of operations to perform that can be distributed across machines in a cluster.

The two ways of transforming data in a Dataflow pipeline are `DoFn` and `Transform` steps. A `DoFn` step defines an operation to perform on each object in a collection. For example, we'll query the Natality public data set and the resulting collection will contain dictionary objects. We'll define a `DoFn` operation that uses `sklearn` to create a prediction for each of these dictionary objects and output a new dictionary object. A `Transform` defines an operation to perform on a set of objects, such as performing feature generation to aggregate raw tracking events into user-level summaries. These types of operations are typically used in combination with a partition transform step to divide up a collection of objects into a manageable size. We won't explore this process in this book, but a transform could be used to apply `Featuretools` to perform automated feature engineering as part of a Dataflow pipeline.

In this chapter we'll get hands on with building Dataflow pipelines that can run locally and in a fully-managed GCP cluster. We'll start by building a simple pipeline that works with text data, and then build a pipeline that applies a `sklearn` model in a distributed workflow.

7.1 Apache Beam

Apache Beam is an open-source library for building data processing workflows using Java, Python, and Go. Beam workflows can be executed across several execution engines including Spark, Dataflow, and MapReduce. With Beam, you can test workflows locally using the `Direct Runner` for execution, and then deploy the workflow in GCP using the `Dataflow Runner`. Beam pipelines can be batch, where a workflow is executed until it is completed, or streaming, where the pipeline runs continuously and operations are performed in near real-time as data is received. We'll focus on batch pipelines in this chapter and cover streaming pipelines in the next chapter.

The first thing we'll need to do in order to get up and running is install the Apache Beam library. Run the commands shown below from the command line in order to install the library, set up credentials for GCP, and to run a test pipeline locally. The `pip` command includes the `gcp` annotation to specify that the Dataflow modules should also be installed. If the last step is successful, the pipeline will output the word counts for Shakespeare's *King Lear*.

```
# install APaChe Bean
pip install --user apache-beam[gcp]

# set up GCP credentials
export GOOGLE_APPLICATION_CREDENTIALS=/home/ec2-user/dsdemo.json

# run the word count example
python3 -m apache_beam.examples.wordcount --output outputs
```

The example pipeline performs a number of different steps in order to perform this counting logic. First, the pipeline reads in the play as a collection of string objects, where each line from the play is a string. Next, the pipeline splits each line into a collection of words, which are then passed to map and group transforms that count the occurrence of each word. The map and group operations are

built-in Bean transform operations. The last step is writing the collection of word counts to the console.

Instead of walking through the example code in detail, we'll build our own pipeline that more closely resembles the workflow of building a batch model application pipeline. The listing below shows the full code for building and running a pipeline that reads in the play from Cloud Storage, appends a message to the end of every line of text, and writes the results back to Cloud Storage. The complete pipeline can be executed from within a Jupyter notebook, which is a useful way of getting up and running with simple pipelines when learning Dataflow.

```
import apache_beam as beam
import argparse
from apache_beam.options.pipeline_options import PipelineOptions
from apache_beam.io import ReadFromText
from apache_beam.io import WriteToText

# define a function for transforming the data
class AppendDoFn(beam.DoFn):
    def process(self, element):
        return element + " - Hello World!"

# set up pipeline parameters
parser = argparse.ArgumentParser()
parser.add_argument('--input', dest='input',
                    default='gs://dataflow-samples/shakespeare/kinglear.txt')
parser.add_argument('--output', dest='output',
                    default='gs://dsp_model_store/shakespeare/kinglear.txt')
known_args, pipeline_args = parser.parse_known_args(None)
pipeline_options = PipelineOptions(pipeline_args)

# define the pipeline steps
p = beam.Pipeline(options=pipeline_options)
lines = p | 'read' >> ReadFromText(known_args.input)
appended = lines | 'append' >> beam.ParDo(AppendDoFn())
```

```
appended | 'write' >> WriteToText(known_args.output)

# run the pipeline
result = p.run()
result.wait_until_finish()
```

The first step in this code is to load the necessary modules needed in order to set up a Beam pipeline. We import IO methods for reading and writing text files, and utilities for passing parameters to the Beam pipeline. Next, we define a class that will perform a `DoFn` operation on every element passed to the `process` function. This class extends the `beam.DoFn` class, which provides an interface for processing elements in a collection. The third step is setting up parameters for the pipeline to use for execution. For this example, we need to set up the input location for reading the text and output location for writing the result.

Once we have set up the pipeline options, we can set up the DAG that defines the sequence of actions to perform. For this example, we'll create a simple sequence where the input text is passed to our append step and the output is passed to the text writer. A visualization of this pipeline is shown in Figure 7.1. To construct the DAG, we use pipe (`|`) commands to chain the different steps together. Each step in the pipeline is a `ParDo` or `Transform` command that defines the Beam operation to perform. In more complicated workflows, an operation can have multiple outputs and multiple inputs.

Once the pipeline is constructed, we can use the `run` function to execute the pipeline. When running this example in Jupyter, the Direct Runner will be used by Beam to execute the pipeline on the local machine. The last command waits for the pipeline to complete before proceeding.

With the Direct Runner, all of the global objects defined in the Python file can be used in the `DoFn` classes, because the code is running as a single process. When using a distributed runner, some additional steps need to be performed to make sure that the class has

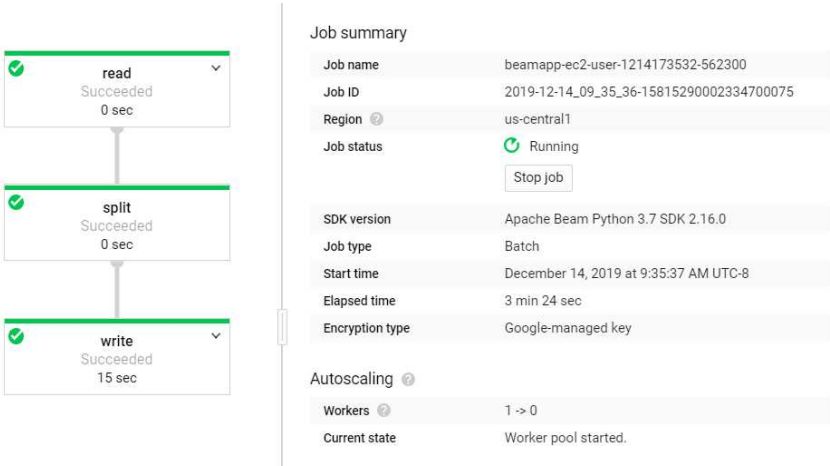


FIGURE 7.1: Running the managed pipeline on GCP.

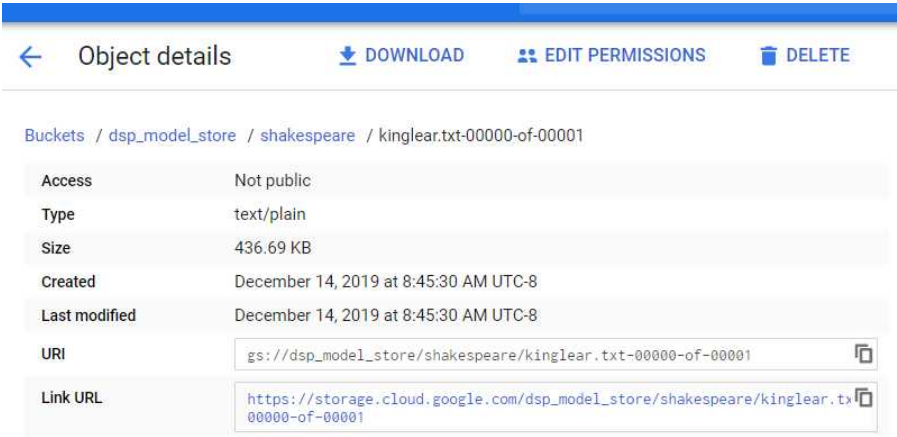


FIGURE 7.2: The resulting file on Google Storage.

access to the modules needed to perform operations. We'll cover this issue in the next section, since the process function in this example does not use any modules. In general, process functions should only make use of modules defined in the `init` function, the passed in elements, and any side inputs that are provided to the model. We won't cover side inputs in this chapter, which provide a way of passing addition data to `DoFn` operations, and instead we'll load the model from Cloud Storage when the class is instantiated.

After running the pipeline, the appended text will be available on Cloud Storage. You can validate that the pipeline was successful by browsing to the bucket in the GCP console, as shown in Figure 7.2. The result is saved as a single file, but for larger outputs the result will be split into multiple files, where the best practice is to use Avro or Parquet formats. The Avro format works well with Dataflow, because data gets streamed between different steps in the pipeline, even when running in batch mode, and results can be written to storage once they are ready on each worker machine. Unlike Spark, where stages with dependencies are not executed concurrently, steps in a Dataflow workflow with dependencies can execute simultaneously.

While it's possible to run pipelines from Jupyter, which is useful for learning how pipelines work, it's more common to use a text editor or IDE to create Python files that are executed via the command line. The first command below shows how to run the example pipeline with the Direct Runner, and the second command shows how to run the pipeline on Cloud Dataflow using the runner parameter. We also need to include a staging location on Cloud Storage for Dataflow to manage the job and specify our GCP project name.

```
# run locally
python3 append.py \

# run managed
python3 append.py \
    --runner DataflowRunner \
    --project your_project_name \
    --temp_location gs://dsp_model_store/tmp/
```

By default, Beam pipelines run as a batch process. To run in streaming mode, you need to pass the streaming flag, which we'll cover in the next chapter. The result of running the workflow on Dataflow is shown in Figure 7.1. You can view the progress of your

workflows on GCP by browsing to the console and navigating to the Dataflow view, which will show a list of jobs.

We now have hands on experience with building a data pipeline using Apache Beam, and have run the pipeline locally and in a managed cloud environment. The next step is to use the `process` function to apply an ML model to the passed in data set.

7.2 Batch Model Pipeline

Cloud Dataflow provides a useful framework for scaling up sklearn models to massive data sets. Instead of fitting all input data into a dataframe, we can score each record individually in the `process` function, and use Apache Beam to stream these outputs to a data sink, such as BigQuery. As long as we have a way of distributing our model across the worker nodes, we can use Dataflow to perform distributed model application. This can be achieved by passing model objects as side inputs to operators or by reading the model from persistent storage such as Cloud Storage. In this section we'll first train a linear regression model using a Jupyter environment, and then store the results to Cloud Storage so that we can run the model on a large data set and save the predictions to BigQuery and Cloud Datastore.

7.2.1 Model Training

The modeling task that we'll be performing is predicting the birth weight of a child given a number of factors, using the Natality public data set. To build a model with sklearn, we can sample the data set before loading it into a Pandas dataframe and fitting the model. The code snippet below shows how to sample the data set from a Jupyter notebook and visualize a subset of records, as shown in Figure 7.3.

	year	plurality	apgar_5min	mother_age	father_age	gestation_weeks	ever_born	mother_married	weight
0	2005	1.0	9.0	34	38	41	10	1	8.628893
1	2005	1.0	6.0	36	39	34	8	1	2.678616
2	2006	1.0	9.0	38	41	41	8	1	11.062796
3	2007	2.0	9.0	42	42	38	8	1	5.436599
4	2007	1.0	8.0	38	43	31	8	1	3.560466

FIGURE 7.3: The sampled Natality data set for training.

```
from google.cloud import bigquery
client = bigquery.Client()

sql = """
SELECT year, plurality, apgar_5min,
       mother_age, father_age,
       gestation_weeks, ever_born
       ,case when mother_married = true
             then 1 else 0 end as mother_married
       ,weight_pounds as weight
FROM `bigquery-public-data.samples.natality`
order by rand()
limit 10000
"""

natalityDF = client.query(sql).to_dataframe().fillna(0)
natalityDF.head()
```

Once we have the data to train on, we can use the `LinearRegression` class in `sklearn` to fit a model. We'll use the full dataframe for fitting, because the holdout data is the rest of the data set that was not sampled. Once trained, we can use `pickle` to serialize the model and save it to disk. The last step is to move the model file from local storage to cloud storage, as shown below. We now have a model trained that can be used as part of a distributed model application workflow.

```

from sklearn.linear_model import LinearRegression
import pickle
from google.cloud import storage

# fit and pickle a model
model = LinearRegression()
model.fit(natalityDF.iloc[:,1:8], natalityDF['weight'])
pickle.dump(model, open("natality.pkl", 'wb'))

# Save to GCS
bucket = storage.Client().get_bucket('dsp_model_store')
blob = bucket.blob('natality/sklearn-linear')
blob.upload_from_filename('natality.pkl')

```

7.2.2 BigQuery Publish

We'll start by building a Beam pipeline that reads in data from BigQuery, applies a model, and then writes the results to BigQuery. In the next section, we'll add Cloud Datastore as an additional data sink for the pipeline. This pipeline will be a bit more complex than the prior example, because we need to use multiple Python modules in the process function, which requires a bit more setup.

We'll walk through different parts of the pipeline this time, to provide additional details about each step. The first task is to define the libraries needed to build and execute the pipeline. We are also importing the `json` module, because we need this to create the schema object that specifies the structure of the output BigQuery table. Like the past section, we are still sampling the data set to make sure our pipeline works before ramping up to the complete data set. Once we're confident in our pipeline, we can remove the `limit` command and autoscale a cluster to complete the workload.

```

import apache_beam as beam
import argparse
from apache_beam.options.pipeline_options import PipelineOptions

```



```

from apache_beam.options.pipeline_options import SetupOptions
from apache_beam.io.gcp.bigquery import parse_table_schema_from_json
import json

query = """
    SELECT year, plurality, apgar_5min,
    mother_age, father_age,
        gestation_weeks, ever_born
        ,case when mother_married = true
            then 1 else 0 end as mother_married
        ,weight_pounds as weight
        ,current_timestamp as time
        ,GENERATE_UUID() as guid
    FROM `bigquery-public-data.samples.natality`
    rand()
    limit 100
    """

```

Next, we'll define a `DoFn` class that implements the `process` function and applies the sklearn model to individual records in the Natality data set. One of the changes from before is that we now have an `init` function, which we use to instantiate a set of fields. In order to have references to the modules that we need to use in the `process` function, we need to assign these as fields in the class, otherwise the references will be undefined when running the function on distributed worker nodes. For example, we use `self._pd` to refer to the Pandas module instead of `pd`. For the model, we'll use lazy initialization to fetch the model from Cloud Storage once it's needed. While it's possible to implement the `setup` function defined by the `DoFn` interface to load the model, there are limitations on which runners call this function.

```

class ApplyDoFn(beam.DoFn):

    def __init__(self):
        self._model = None

```

```

from google.cloud import storage
import pandas as pd
import pickle as pkl
self._storage = storage
self._pkl = pkl
self._pd = pd

def process(self, element):
    if self._model is None:
        bucket = self._storage.Client().get_bucket(
                                                    'dsp_model_store')
        blob = bucket.get_blob('natality/sklearn-linear')
        self._model = self._pkl.loads(blob.download_as_string())

    new_x = self._pd.DataFrame.from_dict(element,
                                         orient = "index").transpose().fillna(0)
    weight = self._model.predict(new_x.iloc[:,1:8])[0]
    return [ { 'guid': element['guid'], 'weight': weight,
               'time': str(element['time']) } ]

```

Once the model object has been lazily loaded in the process function, it can be used to apply the linear regression model to the input record. In Dataflow, records retrieved from BigQuery are returned as a collection of dictionary objects and our process function is responsible for operating on each of these dictionaries independently. We first convert the dictionary to a Pandas dataframe and then pass it to the model to get a predicted weight. The process function returns a list of dictionary objects, which describe the results to write to BigQuery. A list is returned instead of a dictionary, because a process function in Beam can return zero, one, or multiple objects.

An example `element` object passed to process function is shown in the listing below. The object is a dictionary type, where the keys are the column names of the query record and the values are the record values.

```
{'year': 2001, 'plurality': 1, 'apgar_5min': 99, 'mother_age': 33,
  'father_age': 40, 'gestation_weeks': 38, 'ever_born': 8,
  'mother_married': 1, 'weight': 6.8122838958,
  'time': '2019-12-14 23:51:42.560931 UTC',
  'guid': 'b281c5e8-85b2-4cbd-a2d8-e501ca816363'}
```

To save the predictions to BigQuery, we need to define a schema that defines the structure of the predictions table. We can do this using a utility function that converts a JSON description of the table schema into the schema object required by the Beam BigQuery writer. To simplify the process, we can create a Python dictionary object and use the `dumps` command to generate JSON.

```
schema = parse_table_schema_from_json(json.dumps({'fields':
  [ { 'name': 'guid', 'type': 'STRING'},
    { 'name': 'weight', 'type': 'FLOAT64'},
    { 'name': 'time', 'type': 'STRING'} ]}))
```

The next step is to create the pipeline and define a DAG of Beam operations. This time we are not providing input or output arguments to the pipeline, and instead we are passing the input and output destinations to the BigQuery operators. The pipeline has three steps: read from BigQuery, apply the model, and write to BigQuery. To read from BigQuery, we pass in the query and specify that we are using standard SQL. To apply the model, we use our custom class for making predictions. To write the results, we pass the schema and table name to the BigQuery writer, and specify that a new table should be created if necessary and that data should be appended to the table if data already exists.

```
# set up pipeline options
parser = argparse.ArgumentParser()
known_args, pipeline_args = parser.parse_known_args(None)
pipeline_options = PipelineOptions(pipeline_args)
```

Query results [SAVE RESULTS](#) [EXPLORE WITH DATA STUDIO](#)

Query complete (0.6 sec elapsed, 0 B processed)

Job information **Results** JSON Execution details

Row	guid	weight	time
1	105f76ae-9c9c-466b-a61d-c911fcd0a449	7.4475855859655615	2019-12-14 23:47:13.529691 UTC
2	425a31f5-9d01-4ae2-9996-82c27ad56143	7.93343406915932	2019-12-14 23:47:13.529691 UTC
3	dbc71373-7cfd-48e2-8cd3-e6c702587e7e	7.378916852275836	2019-12-14 23:47:13.529691 UTC
4	a4b3bb7d-1d8c-420f-9a23-063d6ff184f7	8.18444827748126	2019-12-14 23:47:13.529691 UTC
5	d8119fe0-cd4c-41f2-b13c-23bdfca302a2	7.699117579767197	2019-12-14 23:47:13.529691 UTC
6	f6132482-5883-472c-90e0-dedb4b0bfb12	7.406936963326817	2019-12-14 23:47:13.529691 UTC

FIGURE 7.4: The Natality predictions table on BigQuery.

```
# define the pipeline steps
p = beam.Pipeline(options=pipeline_options)
data = p | 'Read from BigQuery' >> beam.io.Read(
    beam.io.BigQuerySource(query=query, use_standard_sql=True))
scored = data | 'Apply Model' >> beam.ParDo(ApplyDoFn())
scored | 'Save to BigQuery' >> beam.io.Write(beam.io.BigQuerySink(
    'weight_preds', 'dsp_demo', schema = schema,
    create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
    write_disposition=beam.io.BigQueryDisposition.WRITE_APPEND))
```

The last step in the script is running the pipeline. While it is possible to run this complete code listing from Jupyter, the pipeline will not be able to complete because the `project` parameter needs to be passed as a command line argument to the pipeline.

```
# run the pipeline
result = p.run()
result.wait_until_finish()
```

Before running the pipeline on Dataflow, it's a best practice to run the pipeline locally with a subset of data. In order to run the pipeline locally, it's necessary to specify the GCP project as a

command line argument, as shown below. The project parameter is needed to read and write data with BigQuery. After running the pipeline, you can validate that the workflow was successful by navigating to the BigQuery UI and checking for data in the destination table, as shown in Figure 7.4.

To run the pipeline on Cloud Dataflow, we need to pass a parameter that identifies the Dataflow Runner as the execution engine. We also need to pass the project name and a staging location on Cloud Storage. We now pass in a requirements file that identifies the `google-cloud-storage` library as a dependency, and set a cluster size limit using the `max workers` parameter. Once submitted, you can view the progress of the job by navigating to the Dataflow UI in the GCP console, as shown in Figure 7.5.

```
# running locally
python3 apply.py --project your_project_name

# running on GCP
echo '$google-cloud-storage==1.19.0' > reqs.txt
python3 apply.py \
  --runner DataflowRunner \
  --project your_project_name \
  --temp_location gs://dsp_model_store/tmp/ \
  --requirements_file reqs.txt \
  --maxNumWorkers 5
```

We can now remove the limit command from the query in the pipeline and scale the workload to the full dataset. When running the full-scale pipeline, it's useful to keep an eye on the job to make sure that the cluster size does not scale beyond expectations. Setting the maximum worker count helps avoid issues, but if you forget to set this parameter than the cluster size can quickly scale and result in a costly pipeline run.

One of the potential issues with using Python for Dataflow pipelines is that it can take awhile to initialize a cluster, because each worker node will install the required libraries for the job from

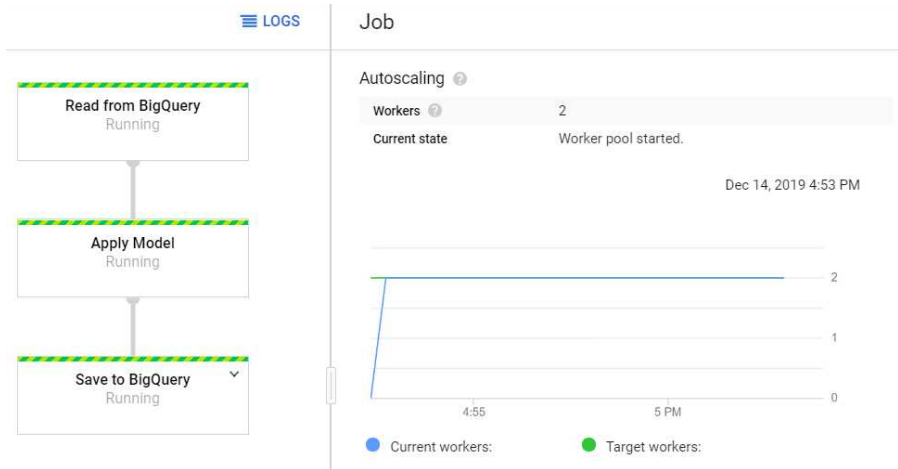


FIGURE 7.5: Running the managed pipeline with autoscaling.

source, which can take a significant amount of time for libraries such as Pandas. To avoid lengthy startup delays, it's helpful to avoid including libraries in the requirements file that are already included in the Dataflow SDK¹. For example, Pandas 0.24.2 is included with SDK version 2.16.0, which is a recent enough version for this pipeline.

One of the useful aspects of Cloud Dataflow is that it is fully managed, which means that it handles provisioning hardware, deals with failures if any issues occur, and can autoscale to match demand. Apache Beam is a great framework for data scientists, because it enables using the same tool for local testing and Google Cloud deployments.

7.2.3 Datastore Publish

Publishing results to BigQuery is useful for ETLs and other applications that are part of batch data pipelines. However, it doesn't work well for use cases where applications need to retrieve a prediction for users with low latency. GCP provides two NoSQL databases that provide a solution for this use case, where you need

¹<https://cloud.google.com/dataflow/docs/concepts/sdk-worker-dependencies>

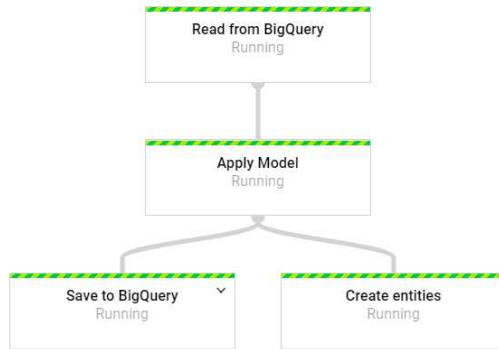


FIGURE 7.6: Publishing to BigQuery and Datastore.

to retrieve information for a specific user with minimal latency. In this section we'll explore Cloud Datastore, which provides some querying capabilities for an application database.

We'll build upon our prior pipeline and add an additional step that publishes to Datastore while also publishing the predictions to BigQuery. The resulting DAG is shown in Figure 7.6. The approach we'll use will write each entity to Datastore as part of the process function. This is much slower than writing all of the entities as a single Beam operation, but the Beam operator that performs this step still requires Python 2.

To update the pipeline, we'll define a new `DoFn` class and add this as the last step in the pipeline, as shown in the code snippet below. The `init` function loads the `datastore` module and makes it referenceable as a field. The `process` function creates a key object that is used to index the entity we want to store, which is similar to a dictionary object. A Datastore entity is the base object used to persist state in a Datastore database. We'll use the `guid` as a unique index for the entity and assign `weight` and `time` attributes to the object. Once we've set the attributes, we use the `put` command to persist the object to Cloud Datastore. This last step can take some time, which is why it is better to return a collection of entities and perform the `put` step as part of a batch operation, if supported.

Entities			
+ CREATE ENTITY IMPORT EXPORT DELETE			
QUERY BY KIND QUERY BY GQL			
<div>Kind</div> <div>natality-guid</div> <div>FILTER ENTITIES</div>			
<input type="checkbox"/>	Name/ID ↑	time	weight
<input type="checkbox"/>	name=0046cdef-6a0f-4586-86ec-4b995cfc7...	2019-12-15 03:00:06.319496 UTC	7.9434742419056
<input type="checkbox"/>	name=077e7597-797f-4554-a6d3-01de3a37...	2019-12-15 03:00:06.319496 UTC	7.682624427455222
<input type="checkbox"/>	name=08de8886-0cac-4633-ad6e-7b693870...	2019-12-15 03:00:06.319496 UTC	6.454754420832512
<input type="checkbox"/>	name=0f7e9b50-3cc9-4c36-9dde-21122d53...	2019-12-15 03:00:06.319496 UTC	7.803163739120405
<input type="checkbox"/>	name=10018d70-4d3c-4874-b262-58b982f5...	2019-12-15 03:00:06.319496 UTC	7.556760799319306

FIGURE 7.7: The resulting entities in Cloud Datastore.

```
class PublishDoFn(beam.DoFn):

    def __init__(self):
        from google.cloud import datastore
        self._ds = datastore

    def process(self, element):
        client = self._ds.Client()
        key = client.key('natality-guid', element['guid'])
        entity = self._ds.Entity(key)
        entity['weight'] = element['weight']
        entity['time'] = element['time']
        client.put(entity)

scored | 'Create entities' >> beam.ParDo(PublishDoFn())
```

We can now rerun the pipeline to publish the predictions to both BigQuery and Datastore. To validate that the model ran successfully, you can navigate to the Datastore UI in the GCP console and inspect the entities, as shown in Figure 7.7.


```
from google.cloud import datastore
client = datastore.Client()
query = client.query(kind='natality-guid')

query_iter = query.fetch()
for entity in query_iter:
    print(entity)
    break
```

It's also possible to fetch the predictions published to Dataflow using Python. The code snippet above shows how to fetch all of the model predictions and print out the first entity retrieved. A sample entity is shown below, which contains the guid as the unique identifier and additional attributes for the weight and time of the model prediction.

```
<Entity('natality-guid', '0046cdef-6a0f-4586-86ec-4b995cfc7c4e')
  {'weight': 7.9434742419056,
   'time': '2019-12-15 03:00:06.319496 UTC'}>
```

We now have a Dataflow pipeline that can scale to a large data set and output predictions to analytical and application databases. This means that other services can fetch these predictions to personalize products. Batch pipelines are one of the most common ways that I've seen ML models productized in the gaming industry, and Dataflow provides a great framework for enabling data scientists to own more of the model production process.

7.3 Conclusion

Dataflow is a powerful data pipeline tool that enables data scientists to rapidly prototype and deploy data processing workflows that can apply machine learning algorithms. The framework provides a few basic operations that can be chained together to define

complex graphs of workflows. One of the key features of Dataflow is that it builds upon an open source library called Apache Beam that enables the workflows to be portable to other cloud environments.

In this chapter we built a batch model pipeline that fetched data from BigQuery, applied a linear regression model, and then persisted the predictions to BigQuery and Cloud Datastore. In the next chapter we'll explore a streaming version of this pipeline and reuse portions of the current pipeline.

Streaming Model Workflows

Many organizations are now using streaming platforms in order to build real-time data pipelines that transform streams of data and move data between different components in a cloud environment. These platforms are typically distributed and provide fault-tolerance for streaming data. In addition to connecting different systems together, these tools also provide the ability to store records and create event queues.

One of the most popular streaming platforms is Apache Kafka, which is an open-source solution for providing message streaming across public and private clouds. Kafka is a hosted solution that requires provisioning and managing a cluster of machines in order to scale. GCP provides a fully-managed streaming platform called PubSub and AWS provides a managed solution called Kinesis. The best option to use depends on your cloud platform, throughput and latency requirements, and DevOps concerns.

With a streaming platform, you can pass data between different components in a cloud environment, as well as external systems. For example, many game companies are now using these platforms to collect gameplay events from mobile games, where an event is transmitted from the game client to a game server, and then passed to the data platform as a stream. The message producer in this case is the game server passing the message to the consumer, which is the data platform that transforms and stores the event.

The connection to data science in production is that streaming platforms can be used to apply ML models as a transform step in a streaming pipeline. For example, you can set up a Python process that reads in messages from a topic, applies a sklearn model, and outputs the prediction to a new topic. This process can be part of

a larger workflow that provides real-time ML predictions for users, such as item recommendations in a mobile game. For the model application step to scale to large volumes of messages, we'll need to use distributed systems such as Spark and Cloud Dataflow rather than a single Python process.

While the model application step in a streaming model pipeline is similar to setting up a Lambda or Cloud Function, which already provides near real-time predictions, a key difference is the ease of integrating with other components in the cloud platform. For example, with Cloud Dataflow you can route the event to BigQuery for storage as well as the model application step, which may push the output to a new message consumer. Another benefit is that it enables using distributed tools such as PySpark to handle requests for model application, versus the endpoint based approaches that service requests in isolation.

One of the benefits of using messaging systems in a cloud platform is that it enables different tools and different programming languages to communicate using standardized interfaces. We'll focus on Python and PySpark in this book, but Java, Go, and many other languages are supported by these platforms. In this chapter, we'll first use Apache Kafka to pass messages between different Python processes and then consume, transform, and produce new messages using PySpark Streaming. Next, we'll use PubSub on GCP to provide near real-time model predictions using Cloud Dataflow in streaming mode.

8.1 Spark Streaming

Streaming data sets have been supported in Spark since version 0.7, but it was not until version 2.3 that a low-latency mode called Structured Streaming was released. With structured streaming, continuous processing can be used to achieve millisecond latencies when scaling to high-volume workloads. The general flow with structured streaming is to read data from an input stream, such

as Kafka, apply a transformation using Spark SQL, Dataframe APIs, or UDFs, and write the results to an output stream. Spark Streaming also works with managed streaming platforms including PubSub and Kinesis, and other frameworks in the Apache ecosystem including Flume.

In this section, we'll first set up a Kafka instance and then produce and consume messages on the same machine using Kafka. Next, we'll show how to consume messages from Kafka using the `read-stream` function in PySpark, and then build a streaming pipeline that applies a sklearn model.

8.1.1 Apache Kafka

Kafka is an open-source streaming platform that was incubated at LinkedIn. It is designed to handle real-time data streams that are high throughput and low latency. It is written in Java and Scala, but supports a range of programming languages for producing and consuming streams through standardized APIs. The platform can scale to large data sets by using horizontal scaling and partitioning to distribute workloads across a cluster of servers called brokers. While open-source Kafka is a hosted solution for message streaming, some cloud providers now offer fully-managed versions of Kafka, such as Amazon's MSK offering.

To show how Kafka can be integrated into a streaming workflow, we'll use a single-node setup to get up and running. For a production environment, you'll want to set up a multi-node cluster for redundancy and improved latency. Since the focus of this chapter is model application, we won't dig into the details of setting up Kafka for high-availability, and instead recommend managed solutions for small teams getting started. To install Kafka, it's useful to browse to the website¹ and find the most recent release. In order to install Kafka, we first need to install Java, and then download and extract the Kafka release. The steps needed to set up a single-node Kafka instance on an EC2 machine are shown in the

¹<https://kafka.apache.org/quickstart>

snippet below. We'll also install a library for working with Kafka in Python called `kafka-python`.

```
sudo yum install -y java
pip install --user kafka-python
wget http://mirror.reverse.net/pub/apache/kafka/2.4.0/
                                                    kafka_2.12-2.4.0.tgz

tar -xzf kafka_2.12-2.4.0.tgz
cd kafka_2.12-2.4.0
bin/zookeeper-server-start.sh config/zookeeper.properties

# new terminal
bin/kafka-server-start.sh config/server.properties

# new terminal
bin/kafka-topics.sh --create --bootstrap-server localhost:9092
                    --replication-factor 1 --partitions 1 --topic dsp

# output
[2019-12-18 10:50:25] INFO Log partition=dsp-0, dir=/tmp/kafka-logs
    Completed load of log with 1 segments, log start offset 0 and
    log end offset 0 in 56 ms (kafka.log.Log)
```

When setting up Kafka, we'll need to spawn three separate processes to run dependencies, start the Kafka service, and create a new topic for publishing messages. The snippet above runs the following processes:

- **Zookeeper:** An Apache project that provides configuration and service discovery for distributed systems.
- **Kafka** Launches the bootstrap service that enables setting up Kafka topics and using streaming APIs.
- **Topics:** Creates a new topic called “dsp”.

The `zookeeper` and `kafka` tasks are long-running processes that will continue to execute until terminated, while the `topics` process will shutdown once the new Kafka topic is set up. The output at the bottom of the snippet shows the output from running this com-

mand, which will be displayed in terminal running the `Kafka` process. In this configuration, we are setting up a single partition for the topic with no replication. We now have a single-node `Kafka` cluster set up for testing message streaming.

The first API that we'll explore is the `Producer` API, which enables processes to publish a message to a topic. To publish a message to our `Kafka` server, we create a producer object by passing in an IP address and a serialization function, which specifies how to encode Python objects into strings that can be passed to the `Kafka` server. The Python snippet below shows how to create the producer and send a dictionary object as a message to the server, publishing the message to the `dsp` topic. The `dict` object contains `hello` and `time` keys. If we run this code, the message should be successfully transmitted to the server, but there will not yet be a consumer to process the message.

```
from kafka import KafkaProducer
from json import dumps
import time

producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                        value_serializer=lambda x: dumps(x).encode('utf-8'))

data = {'hello' : 'world', 'time': time.time()}
producer.send('dsp', data)
```

To set up a process for consuming the message, we'll explore the `consumer` API, which is used to read in streams of data. The Python snippet below shows how to create a consumer object that connects to the `Kafka` server and subscribes to the `dsp` topic. The consumer object returned is iterable and can be used in combination with a `for` loop in order to process messages. In the example below, the `for` loop will suspend execution until the next message arrives and continue iterating until the process is terminated. The `value` object will be a Python dictionary that we passed from the producer, while the `deserializer` function defines how to transform strings to

Python objects. This approach works fine for small-scale streams, but with a larger data volume we also want to distribute the message processing logic, which we'll demonstrate with PySpark in the next section.

```
from kafka import KafkaConsumer
from json import loads

consumer = KafkaConsumer('dsp',
    bootstrap_servers=['localhost:9092'],
    value_deserializer=lambda x: loads(x.decode('utf-8')))

for x in consumer:
    print(x.value)
```

Now that we have Python scripts for producing and consuming messages, we can test message streaming with Kafka. First, run the Consumer script in a Jupyter notebook, and then run the Producer script in a separate notebook. After running the producer cell multiple times, you should see output from the consumer cell similar to the results shown below.

```
{'hello': 'world', 'time': 1576696313.876075}
{'hello': 'world', 'time': 1576696317.435035}
{'hello': 'world', 'time': 1576696318.219239}
```

We can now use Kafka to reliably pass messages between different components in a cloud deployment. While this section used a test configuration for spinning up a Kafka service, the APIs we explored apply to production environments with much larger data volumes. In the next section, we'll explore the `Streams` API which is used to process streaming data, such as applying an ML model.

8.1.2 Sklearn Streaming

To build an end-to-end streaming pipeline with Kafka, we'll leverage Spark streaming to process and transform data as it arrives.

The structured streaming enhancements introduced in Spark 2.3 enable working with dataframes and Spark SQL while abstracting away many of the complexities of dealing with batching and processing data sets. In this section we'll set up a PySpark streaming pipeline that fetches data from a Kafka topic, applies a sklearn model, and writes the output to a new topic. The entire workflow is a single DAG that continuously runs and processes messages from a Kafka service.

In order to get Kafka to work with Databricks, we'll need to edit the Kafka configuration to work with external connections, since Databricks runs on a separate VPC and potentially separate cloud than the Kafka service. Also, we previously used the bootstrap approach to refer to brokers using `localhost` as the IP. On AWS, the Kafka startup script will use the internal IP to listen for connection, and in order to enable connections from remote machines we'll need to update the configuration to use the external IP, as shown below.

```
vi config/server.properties
advertised.listeners=PLAINTEXT://{external_ip}:9092
```

After making this configuration change, you'll need to restart the Kafka process in order to receive inbound connections from Databricks. You'll also need to enable inbound connections from remote machines, by modifying the security group, which is covered in Section 1.4.1. Port 9092 needs to be open for the Spark nodes that will be making connections to the Kafka service.

We'll also set up a second topic, which is used to publish the results of the model application step. The PySpark workflow we will set up will consume messages from a topic, apply a sklearn model, and then write the results to a separate topic, called `preds`. One of the key benefits of this workflow is that you can swap out the pipeline that makes predictions without impacting other components in the system. This is similar to components in a cloud workflow calling out to an endpoint for predictions, but instead of changing the configuration of components calling endpoints to point to new

endpoints, we can seamlessly swap in new backend logic without impacting other components in the workflow.

```
bin/kafka-topics.sh --create --bootstrap-server localhost:9092
                        --replication-factor 1 --partitions 1 --topic preds
```

It's a good practice to start with a basic workflow that simply consumes messages before worrying about how to build out a predictive modeling pipeline, especially when working with streaming data. To make sure that we've correctly set up Kafka for remote connections with Databricks, we can author a minimal script that consumes messages from the stream and outputs the results, as shown in the PySpark snippet below. Databricks will refresh the output on a regular interval and show new data in the output table as it arrives. Setting the `startingOffsets` value to `earliest` means that we'll backload data from the last Kafka checkpoint. Removing this setting will mean that only new messages are displayed in the table.

```
df = spark .readStream.format("kafka")
           .option("kafka.bootstrap.servers", "{external_ip}:9092")
           .option("subscribe", "dsp")
           .option("startingOffsets", "earliest").load()
display(df)
```

Getting Databricks to communicate with the Kafka service can be one of the main challenges in getting this sample pipeline to work, which is why I recommend starting with a minimal PySpark script. It's also useful to author simple UDFs that process the `value` field of the received messages to ensure that the decoded message in PySpark matches the encoded data from the Python process. Once we can consume messages, we'll use a UDF to apply a sklearn model, where UDF refers to a Python function and not a Pandas UDF. As a general practice, it's good to add checkpoints to a Spark workflow, and the snippet above is a good example for checking if the data received matches the data transmitted.

For the Spark streaming example, we'll again use the Games data set, which has ten attributes and a label column. In this workflow, we'll send the feature vector to the streaming pipeline as input, and output an additional prediction column as the output. We'll also append a unique identifier, as shown in the Python snippet below, in order to track the model applications in the pipeline. The snippet below shows how to create a Python `dict` with the ten attributes needed for the model, append a GUID to the dictionary, and send the object to the streaming model topic.

```
from kafka import KafkaProducer
from json import dumps
import time
import uuid

producer = KafkaProducer(bootstrap_servers= ['{external_ip}:9092'],
                        value_serializer=lambda x: dumps(x).encode('utf-8'))

data = { 'G1': 1, 'G2': 0, 'G3': 0, 'G4': 0, 'G5': 0,
        'G6': 0, 'G7': 0, 'G8': 0, 'G9': 0, 'G10': 0,
        'User_ID': str(uuid.uuid1())}
result = producer.send('dsp', data)
result.get()
```

To implement the streaming model pipeline, we'll use PySpark with a Python UDF to apply model predictions as new elements arrive. A Python UDF operates on a single row, while a Pandas UDF operates on a partition of rows. The code for this pipeline is shown in the PySpark snippet below, which first trains a model on the driver node, sets up a data sink for a Kafka stream, defines a UDF for applying an ML model, and then publishes the scores to a new topic as a pipeline output.

```
from pyspark.sql.types import StringType
import json
import pandas as pd
```

```

from sklearn.linear_model import LogisticRegression

# build a logistic regression model
gamesDF = pd.read_csv("https://github.com/bgweber/Twitch/raw/
                      master/Recommendations/games-expand.csv")
model = LogisticRegression()
model.fit(gamesDF.iloc[:,0:10], gamesDF['label'])

# define the UDF for scoring users
def score(row):
    d = json.loads(row)
    p = pd.DataFrame.from_dict(d, orient = "index").transpose()
    pred = model.predict_proba(p.iloc[:,0:10])[0][0]
    result = {'User_ID': d['User_ID'], 'pred': pred }
    return str(json.dumps(result))

# read from Kafka
df = spark.readStream.format("kafka")
    .option("kafka.bootstrap.servers", "{external_ip}:9092")
    .option("subscribe", "dsp").load()

# select the value field and apply the UDF
df = df.selectExpr("CAST(value AS STRING)")
score_udf = udf(score, StringType())
df = df.select( score_udf("value").alias("value"))

# Write results to Kafka
query = df.writeStream.format("kafka")
    .option("kafka.bootstrap.servers", "{external_ip}:9092")
    .option("topic", "preds")
    .option("checkpointLocation", "/temp").start()

```

The script first trains a logistic regression model using data fetched from GitHub. The model object is created on the driver node, but is copied to the worker nodes when used by the UDF. The next step is to define a UDF that we'll apply to streaming records in the

pipeline. The Python UDF takes a string as input, converts the string to a dictionary using the `json` library, and then converts the dictionary into a Pandas dataframe. The dataframe is passed to the model object and the UDF returns a string representation of a dictionary object with `User_ID` and `pred` keys, where the prediction value is the propensity of the user to purchase a specific game.

The next three steps in the pipeline define the PySpark streaming workflow. The `readStream` call sets up the connection to the Kafka broker and subscribes to the `dsp` topic. Next, a `select` statement is used to cast the `value` column of streaming records to a string before passing the value to the UDF, and then creating a new dataframe using the result of the Python UDF. The last step writes the output dataframe to the `preds` topic, using a local directory as a checkpoint location for Kafka. These three steps run as part of a continuous processing workflow, where the steps do not complete, but instead suspend execution until new data arrives. The result is a streaming DAG of operations that processes data as it arrives.

When running a streaming pipeline, Databricks will show details about the workflow below the cell, as shown in Figure 8.1. The green icon identifies that this is a streaming operation that will continue to execute until terminated. There are also charts that visualize data throughput and latency. For a production pipeline, it's useful to run code using orchestration tools such as Airflow with the Databricks operator, but the notebook environment does provide a useful way to run and debug streaming pipelines.

Now that we are streaming model predictions to a new topic, we'll need to create a new consumer for these messages. The Python snippet below shows to consume messages from the broker for the new predictions topic. The only change from the prior consumer is the IP address and the `deserializer` function, which no longer applies an encoding before converting the string to a dictionary.

```
from kafka import KafkaConsumer
from json import loads
```

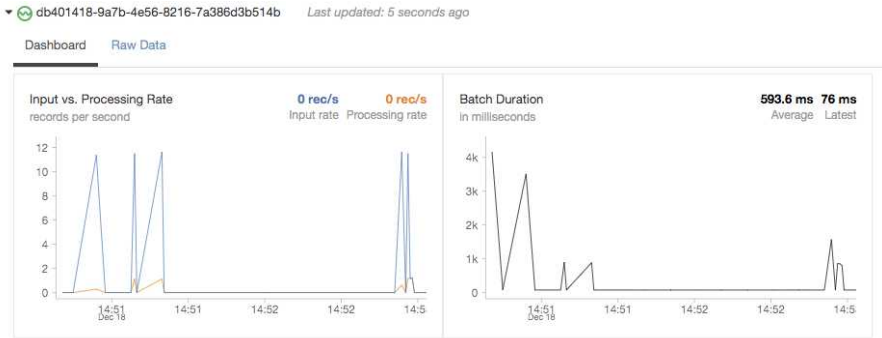


FIGURE 8.1: Visualizing stream processing in Databricks.

```
consumer = KafkaConsumer('preds',
    bootstrap_servers=['{external_ip}:9092'],
    value_deserializer=lambda x: loads(x))

for x in consumer:
    print(x.value)
```

We now have everything in place in order to test out the streaming pipeline with Spark streaming and Kafka. First, run the PySpark pipeline in a Databricks cell. Next, run the consumer script in a Jupyter notebook. To complete the workflow, run the producer script in a separate Jupyter notebook to pass a message to the pipeline. The result should be a prediction dictionary printed to the console of the consumer notebook, as shown below.

```
{'User_ID': '4be94cd4-21e7-11ea-ae04-8c8590b3eee6',
 'pred': 0.9325488640736544}
```

We now have a PySpark streaming pipeline that applies model predictions with near real-time performance. There's additional tuning that we can perform to get this latency to within one millisecond, which is useful for a variety of model and web applications. The benefit of using Spark to perform model application is that we can scale the cluster to match demand and can swap in

new pipelines as needed to provide model updates. Spark streaming was initially a bit tricky to get up and running, but the recent enhancements have made it much easier to get working with model application pipelines. In this pipeline we used a simple regression model, but streaming workflows can also be used for deep learning tasks, such as image classification².

8.2 Dataflow Streaming

In Chapter 7 we explored using Cloud Dataflow to create a batch model pipeline, and authored a `DoFn` to apply a sklearn model. Dataflow can also be used to build streaming model pipelines, by setting a configuration flag. One of the features of Dataflow is that many components can be reused across batch and streaming workflows, and in this section we'll use the same model application class that we defined in the last chapter.

When working with Dataflow in streaming mode, you can use a combination of streaming and batch data sources and streaming data sinks when defining a DAG. For example, you can use the approach from the prior section to read from a Kafka stream, apply a model with a `DoFn` function, and write the results to another stream as a data sink. Some of the GCP systems that work with Dataflow, such as BigQuery, can be used as both a batch and streaming data sink.

In this section, we'll build a streaming pipeline with Dataflow that streams in messages from PubSub, applies a sklearn model, and publishes the results to Cloud Datastore. This type of pipeline is useful for updating a user profile based on real-time data, such as predicting if a user is likely to make a purchase.

²<https://www.youtube.com/watch?v=xwQwKW-cerE>

8.2.1 PubSub

PubSub is a fully-managed streaming platform available on GCP. It provides similar functionality to Kafka for achieving high throughput and low latency when handling large volumes of messages, but reduces the amount of DevOps work needed to maintain the pipeline. One of the benefits of PubSub is that the APIs map well to common use cases for streaming Dataflow pipelines.

One of the differences from Kafka is that PubSub uses separate concepts for producer and consumer data sources. In Kafka, you can publish and subscribe to a topic directly, while in PubSub consumers subscribe to subscriptions rather than directly subscribing to topics. With PubSub, you first set up a topic and then create one or more subscriptions that listen on this topic. To create a topic with PubSub, perform the following steps:

1. Browse to the PubSub UI in the GCP Console
2. Click “Create Topic”
3. Enter “natality” for the topic ID, and click “Create Topic”

The result of performing these actions is that we now have a topic called `natality` that we can use for publishing messages. Next, we’ll create a subscription that listens for messages on this topic by performing these steps:

1. Click on Subscriptions in the navigation pane
2. Click “Create Subscription”
3. Assign a subscription ID “dsp”
4. Select the “natality” topic
5. Click “Create”

We now have a topic and subscription set up for streaming messages in a pipeline. Before setting up a Dataflow pipeline, we’ll first create message consumers and producers in Python. The code snippet below shows how to read messages from the subscription using the Google Cloud library. We first create a subscriber client, then set up the subscription and assign a callback function. Unlike the Kafka approach, which returns an iterable object, PubSub uses

a callback pattern where you provide a function that is used to process messages as they arrive. In this example we simply print the `data` field in the message and then acknowledge that the message has been received. The for loop at the bottom of the code block is used to keep the script running, because none of the other commands suspend when executed.

```
import time
from google.cloud import pubsub_v1

subscriber = pubsub_v1.SubscriberClient()
subscription_path = subscriber.subscription_path(
    "your_project_name", "dsp")

def callback(message):
    print(message.data)
    message.ack()

subscriber.subscribe(subscription_path, callback=callback)

while True:
    time.sleep(10)
```

We'll use the same library to create a message producer in Python. The code snippet below shows how to use the Google Cloud library to create a publishing client, set up a connection to the topic, and publish a message to the `dsp` topic. For the producer, we need to encode the message in `utf-8` format before publishing the message.

```
from google.cloud import pubsub_v1

publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path("your_project_name8", "natality")

data = "Hello World!".encode('utf-8')
publisher.publish(topic_path, data=data)
```

To test out the pipeline, first run the consumer in a Jupyter notebook and then run the producer in a separate Jupyter notebook. The result should be that the consumer cell outputs "Hello World" to the console after receiving a message. Now that we have tested out basic functionality with PubSub, we can now integrate this messaging platform into a streaming Dataflow pipeline.

8.2.2 Natality Streaming

PubSub can be used to provide data sources and data sinks within a Dataflow pipeline, where a consumer is a data source and a publisher is a data sink. We'll reuse the Natality data set to create a pipeline with Dataflow, but for the streaming version we'll use a PubSub consumer as the input data source rather than a BigQuery result set. For the output, we'll publish predictions to Datastore and reuse the `publish DoFn` from the previous chapter.

```
import apache_beam as beam
import argparse
from apache_beam.options.pipeline_options import PipelineOptions
from apache_beam.io.gcp.bigquery import parse_table_schema_from_json
import json

class ApplyDoFn(beam.DoFn):

    def __init__(self):
        self._model = None
        from google.cloud import storage
        import pandas as pd
        import pickle as pkl
        import json as js
        self._storage = storage
        self._pkl = pkl
        self._pd = pd
        self._json = js
```

```

def process(self, element):
    if self._model is None:
        bucket = self._storage.Client().get_bucket(
            'dsp_model_store')
        blob = bucket.get_blob('natality/sklearn-linear')
        self._model = self._pkl.loads(blob.download_as_string())

    element = self._json.loads(element.decode('utf-8'))
    new_x = self._pd.DataFrame.from_dict(element,
                                         orient = "index").transpose().fillna(0)
    weight = self._model.predict(new_x.iloc[:,1:8])[0]
    return [ { 'guid': element['guid'], 'weight': weight,
              'time': str(element['time']) } ]

```

The code snippet above shows the function we'll use to perform model application in the streaming pipeline. This function is the same as the function we defined in Chapter 7 with one modification, the `json.loads` function is used to convert the passed in string into a dictionary object. In the previous pipeline, the elements passed in from the BigQuery result set were already dictionary objects, while the elements passed in from the PubSub consumer are string objects. We'll also reuse the `DoFn` function the past chapter which publishes elements to Datastore, listed in the snippet below.

```

class PublishDoFn(beam.DoFn):

    def __init__(self):
        from google.cloud import datastore
        self._ds = datastore

    def process(self, element):
        client = self._ds.Client()
        key = client.key('natality-guid', element['guid'])
        entity = self._ds.Entity(key)
        entity['weight'] = element['weight']

```

```
entity['time'] = element['time']
client.put(entity)
```

Now that we have defined the functions for model application and publishing to Datastore, we can build a streaming DAG with dataflow. The Python snippet below shows how to build a Dataflow pipeline that reads in a message stream from the *natality* subscription, applies the model application function, and then publishes the output to the application database.

```
# set up pipeline parameters
parser = argparse.ArgumentParser()
known_args, pipeline_args = parser.parse_known_args(None)
pipeline_options = PipelineOptions(pipeline_args)

# define the topics
topic = "projects/{project}/topics/{topic}"
topic = topic.format(project="your_project_name", topic="natality")

# define the pipeline steps
p = beam.Pipeline(options=pipeline_options)
lines = p | 'Read PubSub' >> beam.io.ReadFromPubSub(topic=topic)
scored = lines | 'apply' >> beam.ParDo(ApplyDoFn())
scored | 'Create entities' >> beam.ParDo(PublishDoFn())

# run the pipeline
result = p.run()
result.wait_until_finish()
```

The code does not explicitly state that this is a streaming pipeline, and the code above can be executed in a batch or streaming mode. In order to run this pipeline as a streaming Dataflow deployment, we need to specify the streaming flag as shown below. We can first test the pipeline locally before deploying the pipeline to GCP. For a streaming pipeline, it's best to use GCP deployments, because the fully-managed pipeline can scale to match demand, and the

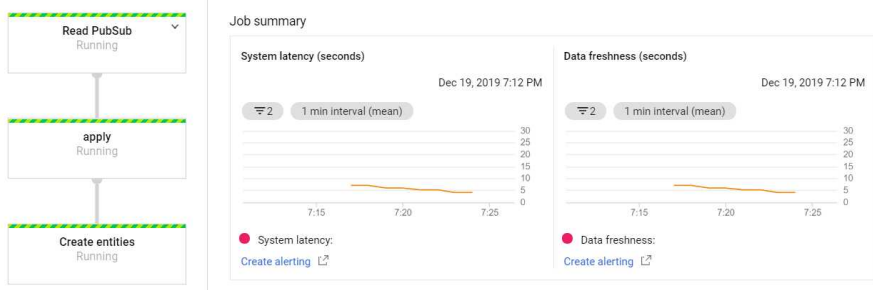


FIGURE 8.2: The Dataflow DAG with streaming metrics.

platform will handle provisioning hardware and provide fault tolerance. A visualization of the Dataflow pipeline running on GCP is shown in Figure 8.2.

```
python3 natality.py --streaming
```

To test out the pipeline, we'll need to pass data to the `dsp` topic which is forwarded to the `natality` subscription. The code snippet below shows how to pass a dictionary object to the topic using Python and the Google Cloud library. The data passed to PubSub represents a single record in the BigQuery result set from the previous chapter.

```
import json
from google.cloud import pubsub_v1
import time

data = json.dumps({'year': 2001, 'plurality': 1,
                  'apgar_5min': 99, 'mother_age': 33,
                  'father_age': 40, 'gestation_weeks': 38, 'ever_born': 8,
                  'mother_married': 1, 'weight': 6.8122838958,
                  'time': str(time.time()),
                  'guid': 'b281c5e8-85b2-4cbd-a2d8-e501ca816363'})
).encode('utf-8')

publisher = pubsub_v1.PublisherClient()
```

<input type="checkbox"/>	Name/ID	time ↑	weight
<input type="checkbox"/>	name=b281c5e8-85b2-4cbd-a2d8-e501ca81...	1576810954.518128	7.700608304146377

FIGURE 8.3: The prediction output pushed to Cloud Datastore.

```
topic_path = publisher.topic_path("your_project_name", "natality")
publisher.publish(topic_path, data=data)
```

The result of passing data to the topic should be an updated entry in Datastore, which provides a model prediction for the passed in GUID. As more data is passed to the pipeline, additional entries will be added to the data set. A sample output of this pipeline is shown in Figure 8.3, which displays the predicted weight for one of the records passed to the pipeline. To run the pipeline on GCP, run the following statement on the command line.

```
python3 natality.py --streaming
--runner DataflowRunner \
--project your_project_name \
--temp_location gs://dsp_model_store/tmp/ \
```

We now have a Dataflow streaming pipeline running in a fully-managed environment. We can use PubSub to interface the pipeline with other components in a cloud deployment, such as a data platform that receives real-time data from mobile applications. With Dataflow, many components can be reused across batch and streaming model pipelines, which makes it a flexible tool for building production pipelines. One factor to consider when using Dataflow for streaming pipelines is that costs can be much

larger when using streaming versus batch operations, such as writing to BigQuery³.

8.3 Conclusion

Streaming model pipelines are useful for systems that need to apply ML models in real-time. To build these types of pipelines, we explored two message brokers that can scale to large volumes of events and provide data sources and data sinks for these pipelines. Streaming pipelines often constrain the types of operations you can perform, due to latency requirements. For example, it would be challenging to build a streaming pipeline that performs feature generation on user data, because historic data would need to be retrieved and combined with the streaming data while maintaining low latency. There are patterns for achieving this type of result, such as precomputing aggregates for a user and storing the data in an application database, but it can be significantly more work getting this type of pipeline to work in a streaming mode versus a batch mode.

We first explored Kafka as a streaming message platform and built a real-time pipeline using structure streaming and PySpark. Next, we built a steaming Dataflow pipeline reusing components from the past chapter that now interface with the PubSub streaming service. Kafka is typically going to provide the best performance in terms of latency between these two message brokers, but it takes significantly more resources to maintain this type of infrastructure versus using a managed solution. For small teams getting started, PubSub or Kinesis provide great options for scaling to match demand while reducing DevOps support.

³<https://labs.spotify.com/2017/10/16/>

8.4 Thank You

Writing this book has been a great experience for getting hands-on with many of the tools that I advocate for data scientists to learn and add to their toolbox. Thank you for reading through this book to the end.

Given the breadth of tools covered in this book, I wasn't able to provide much depth on any particular topic. The next step for readers is to choose a topic introduced in this text and find resources to learn about the topic in more depth than can be covered here.

Data science is a rapidly evolving field, and that means that the contents of this book will become outdated as cloud platforms evolve and libraries are updated. While I was authoring this text, substantial updates were released for TensorFlow which impacted the later chapters. The takeaway is that keeping up with data science as a discipline requires ongoing learning and practice.

Bibliography

- Chollet, F. (2017). *Deep Learning with Python*. Manning, 1st edition. ISBN 978-1617294433.
- Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, 1st edition. ISBN 978-1491962299.
- Grus, J. (2015). *Data Science from Scratch*. O'Reilly Media, 1st edition. ISBN 978-1491901427.
- Karau, H., Konwinski, A., Wendell, P., and Zaharia, M. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 1st edition. ISBN 978-1449358624.
- McKinney, W. (2017). *Python for Data Analysis*. O'Reilly Media, 2nd edition. ISBN 978-1491957660.
- Weber, B. (2018). *Data Science for Startups*. Kindle Direct, 1st edition. ISBN 978-1983057977.
- Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.
- Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.16.

